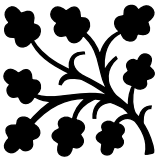
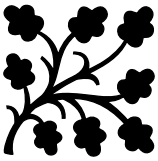
THE PYTHON BOOK

The learning book for beginners of the Institute of Astronomy and Geophysics



The Python Book



This learning book was written in 2021

set in Ulaanbaatar at the Institute of Astronomy and Geophysics

by

Jérémy Hraman

Жереми Храман

2021

Special thanks to:

- my three students Dagzinmaa Lkhagva, Ganbat Baigalaa and Lkhagva Tungalag for attending every single lesson I gave during the year.

Cover

- Jérémy Hraman: Namnaa Archery Horse Riding Club

jeremy.hraman@gmail.com

Contents

1. Discover Python

This chapter is only about theory but it will give you some knowledge about Python, its history and how to install it.

1.1. Computer and Programming

You know for sure that computer communicate with the binary language (00111000110101 for example means absolutely nothing). And for you to interact with your computer, it would be lame if you have to give him instruction with only 0 and 1. Imagine the time you'll need just to close the internet window here. That's why humanity decided to get some programming language, to simplify the writing and the communication with computer. It will translate for you in binary what you want to do.

1.2. Python discovery

1.2.1 History

Python was invented in 1991 (yeah 30 years ago!) by Guido van Rossum who worked first on a Macintosh version until the creation of the Python Software Foundation in 2001. The name comes from the famous humorists *Monty Python*.

1.2.2 An interpreted language

In comparison with others languages like C/C++ which are compiled languages, Python is interpreted. This means that the code you write is executed in live when you push the *Run* button. Compiled languages need to be translate into your computer language before be executed. So, everytime you make a little change, you have to compile your whole script again before launching it.

Advantages are that Python code is easier to run, universal (that's not totally true but you can give your code to a Linux, MacOS user without be disturbed with the

compatibility).

Drawbacks are that it's faster to run a compiled software than a Python software even if with the growing of computer speed, it tends not to be faster anymore. And also you have to install the right version of Python everywhere.

For your own knowledge, in terms of speed : Matlab is the slowest, Python faster and Julia the fastest. We're not here to discuss about Julia but perhaps next year ...

1.2.3 Uses of Python

Python is powerful for several reasons. He's easy to learn, and when you acquired the basis, you can create a lot of things really fast. And to help you, you have a ton of libraries already written and always updated. The power of this language is also found in the number of users. As it's well-known and well-spread around the world, you have also a big community to ask questions and already a lot of answers for literally anything.

With Python you can create:

1. Little *script* to accomplish a specific task on your computer
2. Whole software like games, text editor, audio software, mail service
3. Suite of software which work altogether

With the libraries, you can :

1. Create graphical interface
2. Do some networking
3. Speak with you OS
4. Work with seismic data...

I'm not here to give you all the knowledge, but once you understand how it works, with some small research and time, you can go further alone. Yeah I speak of letting my students alone and it's only fifteen minutes we began... children grow up so fast.

1.2.4 Versions of Python

There are two major versions of Python currently : Python 2 and Python 3. All the Linux OS contain at least Python 2. But since 2020, Python 2 is deprecated. That means that there won't be any updates anymore. Oh no, what will we do? Learn Python 3 directly. There are some changes between these two versions because a lot of things were useless in Python 2 and the writing sometimes confusing, not *clean* enough.

So if you have a script written with Python 2, you have to adapt it to Python 3 to run it correctly. That's not the case between versions of Python 3 (like Python 3.5 and Python 3.9). Your code will always be understandable if you use a a more recent version.

1.2.5 Let's go install it !

Windows

First of all, go on the website:

https://www.python.org/downloads/

And click on *Download Python 3.9.1*. Then install it, follow instructions and you'll see in the Start Menu: Python 3.9.

MacOS

Like Windows, first of all, go on the website:

https://www.python.org/downloads/

And click on *Download Python 3.9.1*. Open the .DMG file and then click on PYTHON.MPKG. Follow instructions and it's done.

Linux my friend

The last stable version is here:

https://www.python.org/ftp/python/3.9.1/Python-3.9.1.tgz

Then, open a terminal and CD in the folder you download it.

1. Unzip the tarball with the command `sudo tar -xzf Python-3.9.1.tgz`. Perhaps it will differ from a Linux version to another.

2. `cd Python-3.9.1`

3. Execute the script configure writing `sudo ./configure` and wait.

4. Compile Python 3.9 writing `sudo make` and then `sudo make install`

1.2.6 PyCharm

We will use a big software to code in Python which is PyCharm. For the beginning, we could use just the command line that we installed just before but we will take directly good habits.

Windows

You can download with this link:

https://download.jetbrains.com/python/pycharm-community-2020.3.3.exe

And then install it.

MacOS

You can download with this link:

https://download.jetbrains.com/python/pycharm-community-2020.3.3.dmg

And then install it.

Linux

You can download with this link:

https://download.jetbrains.com/python/pycharm-community-2020.3.3.tar.gz

And then:

1. `sudo tar xzf pycharm-community-2020.3.3.tar.gz -C /opt/`
2. `cd /opt/pycharm-community-2020.3.3/bin`
3. `sh pycharm.sh`

Launch PyCharm for the first time and it can take some time. After, for the first steps, the JetBrains website has good tutorial to get started:

https://www.jetbrains.com/help/pycharm/quick-start-guide.html

2. Understanding Python and its grammar

2.1. Variables

Now you know what is Python, we can get into it. We will go easy but I think you'll understand everything because you're smart students!

2.1.1 What is it?

Variables are present in all programming languages. Without it, you cannot code a single script. So it's important because if you cannot save a result to use it later, it'll become annoying really fast.

If you compare the memory of your computer with a wardrobe. Each drawer can store data and some of them are your variables.

2.1.2 How does it work?

We take an example. Imagine your computer is asking your age. He stores it in the variable `age` and after your birthday, he will update it adding one to this variable. If someone want to display your age, he can call `age`. To give the variable a value:

```
1 name_of_your_variable = value
```

Wow so complicated.

Few rules:

1. Name of your variable can only contain letters, capital or not, numbers and underscore
2. Name cannot begin with a number
3. Python is case sensitive so `age` is different from `AGE` and `Age`.

A convention with Python is to write the variables lowercase and replacing spaces with underscore. You can do whatever you want, but if you want that your code will be understandable by everyone, it's good to adopt conventions of writing.

In the interpreter (distinguished by >>> in PyCharm), if you write:

```
1 >>> my_age = 26
```

If you run it in the interpreter, nothing happens, and it's normal but if you call afterwards:

```
1 >>> my_age
2 26
```

You see the output displays what you entered just before. It was saved silently and after, you can get it. You can know do simple operations on it like:

```
1 >>> my_age = my_age + 1
2 >>> my_age
3 27
```

Oh wow, I'm older! And you can also use it to create another variable:

```
1 >>> my_age_x2 = my_age * 2
2 >>> my_age_x2
3 54
4 >>> my_age
5 27
```

It doesn't change our first variable but create another one with a different name. If you use the same name twice, it will erase the first value you wrote in it.

Some words are forbidden in Python because they are used for another purposes. Here's the list of them : `and`, `del`, `from`, `none`, `True`, `as`, `elif`, `global`, `nonlocal`, `try`, `assert`, `else`, `if`, `not`, `while`, `break`, `except`, `import`, `or`, `with`, `class`, `False`, `in`, `pass`, `yield`, `continue`, `finally`, `is`, `raise`, `def`, `for`, `lambda`, `return`.

We will see most of them during this class in the next lessons. Don't be afraid, it seems a lot to learn but when you're used to it, it's really easy.

2.2. Types and Functions

Nothing complicated to understand but Types in Python are a concept which is compulsory to understand if you want to go deeper in Python.

We just saw some numbers before but that's not the only data type which exists in Python and you will even create your own after, but don't skip step! It is useful for Python to know what we work with because when you'll do some operations, the behavior of Python is not the same if you work with numbers or with strings.

We begin with the easiest Data types and there will be two whole chapters with the others.

2.2.1 Integer and Float

Integers are numbers without commas like 3. That's not the same as 3.0 (called **float**) because the calculation method is different. When you add two integers together, there is no error possible. But if you add two floats together, you have a possibility of mistake because your computer cannot store all the numbers behind the comma.

```
1 >>> 3 + 3 # result is an integer
2 6
3 >>> 3.1 + 3 # result is a float
4 6.1
```

One error of approximation can be easily seen with floats:

```
1 >>> 3.1 + 3.2
2 6.300000000000001
```

Why is there this $1 * 10^{-15}$? It's a little complicated but the representation of your float is binary and there are some little mistake of approximation when it's not an even number in binary representation (like 0.3 here). I won't go further with this, it's totally out of subject and it's more how your computer works...

2.2.2 String

Another type you will use a lot is the string. You can store a bunch of letters, a sentence with quotation marks `"Hello, World!"` or with apostrophes `'Hello, World!'` (or

with triple quotation marks but we won't use it for this purpose, it's confusing and useless).

You can also store in variable like everything.

```
>>> my_string = "Hello, World"
>>> my_string
'Hello, World'
```

But what if I also want to put an apostrophe inside of my string? Good question. There is an escaping symbol and it's the backslash \. So if you want to write *I'm here baby*:

```
>>> my_string = 'I\'m here baby'
>>> my_string
"I'm here baby"
```

But if you use quotation marks, it's optional to write the backslash : `my_string = "I'm here baby"` will work fine.

If you don't use the backslash with apostrophes, Python will think that your string ends after I and that you write some random letters after. And it creates an error:

```
>>> my_string = 'I'm here baby'

File "<ipython-input-2-f5e240eab80d>", line 1
    my_string = 'I'm here baby'
                   ^
    SyntaxError: invalid syntax
```

You can also see the color changing which indicates you that something is wrong.

To create a new line in your string, you have to use `\n`. Python will interpret it correctly.

```
>>> my_string = 'Hello\nWorld\n!'
>>> my_string
'Hello\nWorld\n!'
```

Why doesn't it work? Because we want to call the variables directly in the interpreter. Five minutes and you will know how to display it in a good way.

Another thing: you can add strings together like this:

```
>>> first_word = "Hello"
>>> second_word = "World"
>>> first_word + second_word
'HelloWorld'
```

You just see that there is no spaces. If you want it, you have to add it in one of your two strings.

```
>>> first_word = "Hello "
>>> second_word = "World"
>>> first_word + second_word
'Hello World'
```

2.2.3 Boolean

Booleans is the basis for conditional programming. You can test if a condition is `True` or `False`. Python will return these two words that you can also use yourself to affect a variable with booleans. If you write without the uppercase T or F, Python won't understand. It's not a reason to name a variable true or false. It will be confusing!

You have different operands to test the condition:

`<` : strictly inferior

`>` : strictly superior

`<=` : inferior or equal

`>=` : superior or equal

`==` : equal

`!=` : different

```
a = True
b = False

a = true
b = false # incorrect, the words aren't highlighten

    NameError: name 'true' is not defined
```

To use the operands, you just have to compare two things together:

```
>>> 1 != 0
True
```

Be careful, to verify if it's equal, it's `==` and to affect a value to a variable it's `=`. It is one of the most common error when beginning to program.

```
>>> a = 1
>>> a == 1
True
```

The first line affects the value `1` to the variable `a`. It doesn't return anything. The second line compares the variable `a` to the value `1`.

There are some useful little tricks in Python to save time. You remember when I did `age = age + 1`? There is a little trick to go faster because programming is for lazy people! You can do it another way.

```
>>> age = 26
>>> age += 1 # it's the same as age = age + 1
>>> age
27
```

It works also for `-=`, `*=` and `/=` but that's not as usual as `+=`.

Another trick is the permutation, when you want to exchange two values. With other languages, you have to call a third variable to permute but not with Python.

Permutation classic way

```
1 >>> a = 20
2 >>> b = 30
3 >>> c = a # 3rd variable to permute
4 >>> a = b
5 >>> b = c
6 >>> a
7 30
8 >>> b
9 20
```

In Python you can do it directly without calling a third variable in just one line.

Permutation with Python

```
1 >>> a = 20
2 >>> b = 30
3 >>> a, b = b, a # Python permutation
4 >>> a
5 30
6 >>> b
7 20
```

And a last trick is the multiple affectation of variables. If you want to affect multiple variables on the just one line, you just put a comma between variables and after the sign equal between the values. To give the same value to different variables, you can add more equal signs on the same line.

Multiple affectation

```
1  >>> a, b = 20, True # multiple affection
2  >>> a
3  20
4  >>> b
5  True
6  >>> a = b = 20 # multiple affection with same value
7  >>> a
8  20
9  >>> b
10 20
```

2.2.4 Introduction to Functions

I will introduce functions here but there will be in the second part a whole chapter for functions.

Functions are here to execute several instructions (lines of code) just calling its name. You choose its little name and after you can call it as often as you want without write all the instructions inside.

Most of the functions need at least one parameter to work. These parameters allow the function to work directly on the data you enter.

Functions are called with this syntax:

```
1 name_of_function(parameter_1, parameter_2, ...,
    parameter_n)
```

So you begin writing the name of the function and after, you place parenthesis even if there is no parameters.

If it's blurry in your mind. It's because I just explained with words what's better with example.

`print` function

This function allows you to display, in a correct way, what you want. Not just by typing the name of the variable. And when we will work out of the interpreter, just writing the name of the variable won't display it anymore. You remember also that to display a new line, we have to write `\n`. So with `print`, it will work well. The number of parameters inside the function depends on what you want to do!

`print` function

```
1 a = 3
2 print(a)
3 a = a + 3
4 b = a - 2
5 print("a =", a, "and b =", b)
```

```
3
a = 6 and b = 4
```

What's happening Jérémy? Why don't you put >>> anymore?

I will start now to write the Python code like above because we will have more and more output with longer code. Before, I was writing directly in the interpreter of Python and now, it's a script. A script is a file with several lines of codes and when launched, gives all the outputs of the content. So for this example, the two lines of outputs are the displayed values of the `print` function (lines 2 and 5).

You see all the power of `print`. The first call just display the value of the variable `a`. The second call takes four parameters: two strings and two variables separated by a space. Bring back the example of *Hello World!* with new lines between words:

```
1 my_string = 'Hello\nWorld\n!'
2 print(my_string)
```

```
Hello
World
!
```

`type` function

Just before, we see different types of variables and the `type` function returns what ... type is the variable obviously! You just write `type(name_variable)` and you get the result. It also works directly with number without calling a variable :

`type` function

```
1 a = True
2 type(a) # name of the function and parenthesis
3 type(5)
```

```
bool
int
```

`int` is the abbreviation of integer, `float` for floating number, `str` for strings and `bool` for booleans. We call the type of the variable : the **class**. I just say this to you now to do inception in your mind for the third part ...

`round` function

The `round` function is useful to approximate a number. You remember the *error* when adding the two floats `3.1` and `3.2`. The `round` function makes the error disappear. To use it, two parameters are needed: `round(number, nb_digits)`. The first one is the number you want to approximate, the second one is the number of digits after the comma.

`round` function

```
1 a = 3.1 + 3.2
2 round_a = round(3.1 + 3.2, 1)
3 print('Without approximation:', a)
4 print('With approximation:', round_a)
```

```
Without approximation: 6.300000000000001
With approximation: 6.3
```

2.3. Flow Control Statement

We just executed some instructions without too much purposes. But it will be good if we can get some better things to do, or execute something if a condition is filled. That's why we have a conditional statement, like in any other languages also... Perhaps you feel a little confused and where we're going, that's slow but you could at the end of this first part, create a little script to do a specific task.

2.3.1 Conditional Statement

We begin with an example and we will explain every line of it. You already saw it but I write a lot of `# something`. That's a comment and it's ignored by Python. It's just for us, to explain the code, to precise something. Comments are really important because when you will write a script and after three months you have to change something, you will forget all what you knew about it and the comments are here to remind you what was the purpose of a function, of your script.

First example of a conditional statement

```
1 # First example
2 a = -5
3 if a < 0: # condition to verify if a is positive
4     print("a is inferior to 0")
5     #statements block
```

```
a is inferior to 0
```

1. First line is a comment ignored by the interpreter
2. Affect 5 to the variable a
3. Here is the conditional statement :
   (a) `if` keyword to call a condition
   (b) `a < 0` is the condition we test. You can use all the boolean operators we saw before.
   (c) `:` these two points are important and Python will return you an error if you forget them.
4. The **statements block** you want to execute in case a is inferior to 0. It will simply print a is inferior to 0. You see that the text is **indented**.

Two important notions here: the **statements block** and **indentation**. Indentation are compulsory in Python. Else the interpreter won't understand that you are in your statements block.

Full form (`if`, `elif` and `else`)

If we just use `if`, it can be limited. You just ask your script a condition and nothing happens if the condition is not filled. Or you have to write another if statement with another condition... The first word which is useful is `else`. If your condition is not filled, it will execute the `else` statement. If you script enter in the `if`-statements block, it's impossible to execute what's in the `else`-statements block.

When we come again with the example:

`if-else` statement

```
1 a = 0
2 if a > 0:
3     print("a is strictly superior to 0.")
4 else:
5     print("a is inferior or equal to 0.")
```

```
a is inferior or equal to 0.
```

And what if we want to also say that the variable is equal to zero? We will use the keyword `elif` which is a contraction of else if. So our script is now:

`if-elif-else` statement

```
1 a = -4
2 if a > 0:
3     print("a is superior to 0.")
4 elif a < 0:
5     print("a is inferior to 0.")
6 else:
7     print("a is equal to 0.")
```

```
a is inferior to 0.
```

You can write as many `elif` as you want and just one `else` at the end but there are optional like in the first example. There is no `else` or `elif` and the condition works! Nothing happens when the condition is not filled, that's all.

Keywords `and`, `or` and `is`, `not`

A lot of time, you will have to test several conditions to execute statements like verify that a variable is in an interval. With just what we learned:

Interval first version

```
1 # Verify if a is in the interval [2; 8]
2 a = 9
3 if a >= 2:
4     if a <= 8:
5         print("a is in the interval.")
6     else:
7         print("a is not in the interval.")
8 else:
9     print("a is not in the interval.")
```

```
a is not in the interval.
```

But there is the little word called `and` which can accomplish the two conditions in just one line. It will search if `a >= 2` and `a <= 8` faster.

Interval with an `and`

```
1 a = 5
2 if a >= 2 and a <= 8:
3     print("a is in the interval.")
4 else:
5     print("a is not in the interval.")
```

```
a is in the interval.
```

It's better, faster, stronger. You understand better directly when you see it. Another useful word is `or`. If one or more of the conditions are `True`, it will execute the statements block. Applying to our example:

Interval with an `or`

```
1 a = 0
2 if a < 2 or a > 8:
3     print("a is not in the interval.")
4 else:
5     print("a is in the interval.")
```

```
a is not in the interval.
```

The keyword `not` will give the opposite result if you test something. It's clearer when you test directly booleans. It works also for numbers but with the operator `!=` (different to), you have the same result. So `not a == 5` is equivalent to `a != 5`.

Keyword `not`

```
1 adult = False
2 if adult is not True:
3     print("You're a child.")
```

```
You're a child.
```

You also see the little word `is` which will be better for this kind of situation. If we go deeper in Python, `is` is different from `==` but for integer, float and boolean, it's the same. Just remember that.

2.3.2 The Loops

Loops allow you to repeat a task as much as you want. Concept is a little strange at first but you'll understand. We can, for example, get any character from a string...

Firstly, if I ask you to display the multiplication table of 3 from 1×3 to 5×3, how do you proceed?

Table of 3

```
print('1 x 3 =', 1 * 3)
print('2 x 3 =', 2 * 3)
```

```
1 x 3 = 3
2 x 3 = 6
```

Now, if I ask you to write the multiplication table of 4 from 1×4 to 5×4 but smarter using a variable in case I ask you after to write with 5 or 6...

Table of 4

```
nb = 4
print('1 x', nb, '=', nb * 1)
print('2 x', nb, '=', nb * 2)
print('3 x', nb, '=', nb * 3)
print('4 x', nb, '=', nb * 4)
print('5 x', nb, '=', nb * 5)
```

```
1 x 4 = 4
2 x 4 = 8
3 x 4 = 12
4 x 4 = 16
5 x 4 = 20
```

It's smarter but not efficient at all. That's why we'll use the loops!

`while`-loop

This loop execute a **indented statements block** `while` the condition you wrote is **True**.

Syntax of `while`

```
while condition:
    # statement 1
    # statement 2
    #...
```

What kind of condition can we write? In our example of multiplication table of 3 from 1 to 5, we'll increment a variable each time we go in the while-loop until this variable is greater than 5.

Table of 3 with a `while`-loop

```
table_number = 3 # the table of multiplication we want to
    display
i = 0 # initialization of the variable which will be
   incremented

while i < 5: # condition for the variable
    print(i + 1, "*", table_number, "=", (i + 1) *
       table_number)
    i += 1 # we increment our counter
```

```
1 x 3 = 3
2 x 3 = 6
3 x 3 = 9
4 x 3 = 12
5 x 3 = 15
```

Line by line:

1. Affecting the value 3 for the multiplication table
2. That's our counter. We initialize it at 0. Why 0? It's an habit of programming you have to get in Python. You'll understand when we'll see lists. For the people who know about Matlab, lists begin at 1 but in Python and some many other languages, lists begin at 0.
3. Appreciating a line break for readability
4. Our first line of `while` loop with the condition : while our counter is stricly inferior to 5. And don't forget the two points (colon).
5. The indented block begin here with the `print` function. I let you understand.

6. Most important part: we increment the counter. Each time, `i` variable will go from 0 to 1, then 1 to 2, ... until the condition `i < 5` become `False`. If we don't do it, it will be an infinite loop and we execute `while` statements until the user stop it with `Ctrl + C`.

`for`-loop

This statement works on sequences. It's specialized when you have to browse a sequence of multiple datas. One of a sequence we already saw together is the string. A string is a sequence of characters.

Syntax of a `for`-loop

```
for element in sequence:
    # statement 1
    # ...
```

`element` is a variable created by the `for`. You don't have to affect yourself. During the loop, `element` will take successively all the values of the browsing `sequence`. For example with the string *Sain uu?*:

String example

```
string = "Sain uu?"

for letter in string:
    print(letter)
```

```
S
a
i
n

u
u
?
```

Here, no need to increment letter and no condition needed, it will do its job alone. You see here a new keyword: `in`. It's not only used during `for`-loop but it can also used in an `if`-statement.

Keyword `in`

```
string = "Sain uu?"

for letter in string:
    if letter in "AIUaiu":
        print(letter)
    else:
        print("*")
```

```
*
a
i
*
*
u
u
*
```

It verifies if a letter is in the string we write. If it's the case, condition becomes `True` and we execute the statement.

The function `range()` in a `for`-loop can be useful. It will be the role of a counter but in a `for`-loop. You don't have to increment your variable because it's a `for`-loop. Simple example, you want to count from 0 to 5:

`range` function with two parameters

```
for i in range(0, 6):
    print("i =", i)
```

```
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
```

Explanation: `range(0, 6)` create a sequence in which `i` will browse its values. First parameter is the **start number**, second parameter is the **ending number** BUT excluded. So it will go from 0 to 5, not 6. It is possible to add a third parameter which is the **step** between two numbers:

`range` function with three parameters

```
1 for i in range(0, 11, 2):
2     print("i =", i)
```

```
i = 0
i = 2
i = 4
i = 6
i = 8
i = 10
```


New functions `input()` and `int()`

The function `input()` allows the user to write something, to interact with the script. you have to save the result in a variable:

Function `input()`

```
1 year = input("Write a year: ")
2 print("year:", year)
3 print("type of year:", type(year))
```

```
Write a year: 1995
year: 1995
type of year: <class 'str'>
```

Yes, when you use the `input` function, it returns a string... So we have also the function `int` which takes a variable in parameter and returns an integer as result.

Function `int()`

```
1 year = input("Write a year: ")
2 year = int(year)
3 print("year:", year)
4 print("type of year:", type(year))
```

```
Write a year: 1995
year: 1995
type of year: <class 'int'>
```

Keywords `break` and `continue`

The **break** literally stops a loop, even if the condition of the loop is `True`. I use it personally sometimes but if can avoid it with a condition in the while, try to use this solution and not the break one. It's clearer for a external viewer to see what your script does if you don't use the `break`. Little example of a break situation and also how to avoid it.

Solution with a `break`

```
1 while 1: # 1 is always True -> infinite loop
2     letter = input("Write 'Q' to quit: ")
3     if letter == "Q":
4         print("I quit.")
5         break
6
7 print('End of program')
```

```
Write 'Q' to quit: g
Write 'Q' to quit: q
Write 'Q' to quit: Q
I quit.
End of program
```

Solution without a `break`

```
1 answer = True
2 while answer:
3     letter = input("Write 'Q' to quit: ")
4     if letter == "Q":
5         print('I quit.')
6         answer = False
7
8 print('End of program')
```

```
Write 'Q' to quit: d
Write 'Q' to quit: f
Write 'Q' to quit: Q
I quit.
End of program
```

The keyword `continue`, when read by Python, will go back to the while/for statement and continue the work until the condition is not `True` anymore.

Keyword continue

```
1 i = 1
2 while i < 20:
3     if i % 3 == 0:
4         i += 4
5         print("We increment i with 4. i is now equal to",
              i)
6         continue
7
8     print("The variable i =", i)
9     i += 1
```

```
The variable i = 1
The variable i = 2
We increment i with 4. i is now equal to 7
The variable i = 7
The variable i = 8
We increment i with 4. i is now equal to 13
The variable i = 13
The variable i = 14
We increment i with 4. i is now equal to 19
The variable i = 19
```

2.3.3 Exercises

Warmup 1

Print a text which is affected to a variable.

Warmup 2

Transform your age into the good type to print it.

```
1 age = 26
2 text = "You're " + ... + " now!"
```

Warmup 3

Ask a user his name and print it.

Warmup 4

Create a converter that will ask for amount of days and convert it to years, then print it. Using `int()` function, convert the user's answer to integer. And then make your calculation.

Exercise 1

Print `Hello World` if `a` is greater than `b`.

```
1 a = 35
2 b = 25
3 #...
```

Exercise 2

Print `i` as long as `i` is less than 6.

Exercise 3

Loop/Browse through a string. Each time the letter *a* appears, print `Hey, I'm here`.

Exercise 4

Given two integer numbers, print their product. If the product is greater than 1000, then print their sum.

```
a = 15
b = 100
#...
```

Exercise 5

Given a **range** of the first 10 numbers, Iterate from the start number to the end number, and In each iteration print the sum of the current number and previous number.

Exercise 6

Print the following pattern:

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

Exercise 7

Print each digit from an integer in the reverse order (For example 1234 gives 4321). Think about what does `number % 10`...

Exercise 8

Calculate income tax for the given income by adhering to the below rules:

Taxable Income	Rate (in %)
First 500,000T	0
Between 500,000 and 1,000,000T	10
After 1,000,000T	20

```
income = 1500000
#...
```

Exercise 9

Print multiplication table form 1 to 10 like this:

```
1:   1 2 3 4 5 6 7 8 9 10
2:   2 4 6 8 10 12 14 16 18 20
3:   3 6 9 12 15 18 21 24 27 30
4:   4 8 12 16 20 24 28 32 36 40
5:   5 10 15 20 25 30 35 40 45 50
6:   6 12 18 24 30 36 42 48 54 60
7:   7 14 21 28 35 42 49 56 63 70
8:   8 16 24 32 40 48 56 64 72 80
9:   9 18 27 36 45 54 63 72 81 90
10: 10 20 30 40 50 60 70 80 90 100
```

Exercise 10

For a given word, add a star * after every letter of the word and print it at the end. Example: `word = 'python'` gives `p*y*t*h*o*n*`

Exercise 11

Count all letters, digits, and special symbols from a given string.

```
str1 = "P@#yn26at^&i5ve"
#...
```

```
Total counts of chars, digits,and symbols
Chars = 8
Digits = 3
Symbol = 4
```

The method to find letters for a string is: `isalpha()` and to find numbers: `isnumeric()`.

Example:

```
string = "A"
print(string.isalpha())
print(string.isnumeric())
```

```
True
False
```

Exercise 12

Write a script that asks the user to enter their name and their age. Print out a message addressed to them that tells them the year that they will turn 100 years old.

Exercise 13

Ask the user for a number. Depending on whether the number is even or odd, print out an appropriate message to the user.

Exercise 14

Write a script which will print Fizz / Buzz or FizzBuzz from a user given input.

1. If the number is divisible by 3, it should print "Fizz".
2. If it is divisible by 5, it should print "Buzz".
3. If it is divisible by both 3 and 5, it should print "FizzBuzz".
4. Otherwise, it should print the number.

Exercise 15

Make a two-player Rock-Paper-Scissors game. Ask the two players their names. Then ask them what they want to play. After I let you think about it. And to finish, print the result and the winner.

Remember the rules:

1. Rock beats scissors
2. Scissors beats paper
3. Paper beats rock

2.4. Lists & Tuples

I hope you begin to love Python. It's one of the most powerful language but we only begin to touch the head of it. The two next parts about Lists and Dictionaries are really important. You will use at least the lists everywhere.

Lists are sequences which will contain other objects of any types and even mix it inside of it.

2.4.1 Creating and editing our first lists

There are two ways to create a list.

Creation of a list

```
1 my_list = list()
2
3 my_list2 = [] # I personnaly prefer this one
4
5 print('my_list:', type(my_list))
6 print('my_list2:', type(my_list2))
```

```
my_list: <class 'list'>
my_list2: <class 'list'>
```

Jérémy, I see the word class and I panicked. What is this ?

Don't worry, it's not a insult or anything. It's part of Python syntax to create objects. I will write about it with you a little now. If you don't understand, doesn't matter because we will explain it really in a better way in chapter four.

2.4.2 Brief introduction to methods and classes

Here we go with an example: how to lowercase a sentence. The concept is the following one:

Lowercase concept

```
>>> my_str = 'MY FIRST STRING'
>>> lowercase_my_str(my_str)
'my first string'
```

But we don't want to create a function for it, we will use something already written for the **string** class.

Creation of a list

```
my_str = 'MY FIRST STRING'
print( my_str.lower() )
```

```
my first string
```

Oh wow, what's the hell? A point between my variable and a ... function? What's going on?
It's easy. The point between your string `my_str` and the function `lower()` represents the belonging. `lower` belongs to `my_str`. If you write `type(my_str)` you'll receive `<class 'str'>`. Because your **object** `my_str` comes from the **class** `str`. A class is a data type, but it allows to define type-specific functions and variables. We call it **methods** and **attributes**. Remember these names!

So each time you create a **string**, it comes from the **class** `str` and you call **methods** like `lower()` with a point between the **variable** and the **method** to get some functions associated to the **class**.

Be careful with our example, it won't store automatically your new object. See:

lower method for strings

```
1 my_str = 'MY FIRST STRING'
2 print('my_str.lower:', my_str.lower()) # it prints the
    lowered string
3 print('my_str:', my_str) # It is always the first object
    which is printed.
4
5 # Now I store it :
6 my_str = my_str.lower()
7 print('my_str:', my_str) # and print again, now it's
    lowercase!
```

```
my_str.lower: my first string
my_str: MY FIRST STRING
my_str: my first string
```

2.4.3 Go back to the lists

You can initialize some data directly in the list when creating it. You don't have to create first empty. And not only with numbers, but everything you want can be instanced in a list. Even list itself as you see.

Creation of a list with elements

```
1 my_list = [1, 2, 3]
2 print(my_list)
3
4 my_list = ['Hey', 'You', 18, [1, 2, 3]]
5 print(my_list)
```

```
[1, 2, 3]
['Hey', 'You', 18, [1, 2, 3]]
```

You also see that we erase the previous list if we affect again a new value. Now how can we access just one element in a list ? Here we are:

Access to one element of a list

```
1 my_list = ['Hey', 'You', 18, 3.14]
2 print(my_list[2])
```

```
18
```

The second element is `18`? Nope! The second element is `You`! But why it displays something else?
Most important thing in Python, and I said to you already before, lists began at index 0. So if you want to access the first element:

Access to the first element of a list

```
1 my_list = ['Hey', 'You', 18, 3.14]
2 print(my_list[0])
```

```
Hey
```

And now it works! Magic?

It's good to access one element, but it would be even better if we can access a part of it, or even the last element without knowing at the first place the length of our list.

Access to a part of a list

```
1 pi = [3.1, 3.14, 3.142, 3.1416, 3.14159, 3.141593,
     3.1415927] # Here's the approximation of pi
2
3 print(pi[0])
4 print(pi[1:4])
5 print(pi[-1])
```

```
3.1
[3.14, 3.142, 3.1416]
3.1415927
```


1. As with the `range()` function, when you access a part, the last number is excluded.

2. The great artefact to get the last element in Python is the index `-1` and after you can write `pi[-2]` to get the second last element, ...

If you want to change one value in the list, you just have to affect again for the index you want.

Change an element of a list

```
1 pi = [3.1, 3.14, 3.142, 3.1416, 3.14159, 3.141593,
      3.1415927] # Here's the approximation of pi
2
3 print(pi)
4
5 pi[0] = 42
6 print(pi)
```

```
[3.1, 3.14, 3.142, 3.1416, 3.14159, 3.141593, 3.1415927]
[42, 3.14, 3.142, 3.1416, 3.14159, 3.141593, 3.1415927]
```

2.4.4 Insert and remove objects in a existing list

To add some new element to the end of our list, we use the method `append()`. A method ? Yeah remember just before, I said a function used inside an object like `my_string.lower()` is called a method! Let's see:

Insert an element in a list

```
1 dishes = ['buuz', 'khuushuur']
2 print(dishes)
3
4 dishes.append('khorkhog')
5 print(dishes)
```

```
['buuz', 'khuushuur']
['buuz', 'khuushuur', 'khorkhog']
```

Jeremy, why don't we store `my_list` when we add an element like with strings?
Wow you remember so well! It's the most important difference between `list` and `str`. The result of a method will automatically update your list, not with a string.

Comparison of list and string

```
1  my_list = [1, 2, 3]
2  my_list.append(4) # it updates the list
3  print(my_list)
4
5  my_str = "HEY"
6  my_str.lower() # it doesn't update the string
7  print(my_str)
8
9  my_str = my_str.lower() # it update the string
10 print(my_str)
```

```
[1, 2, 3, 4]
HEY
hey
```

And if you try to store a list, you will have a `None` object because the methods of a list don't return anything.

`None` object with method of a list

```
1  my_list = [1, 2, 3]
2  my_list2 = my_list.append(4) # try to store the result in
       my_list2
3  print('my_list: ', my_list)
4  print('my_list2: ', my_list2)
```

```
my_list:  [1, 2, 3, 4]
my_list2:  None
```

If you want to insert on a certain index, you can use `insert(index, value)` method.

To remove some value, easy: `remove(value)`. Be careful, if there is a value several times in the list, it will only remove the first one it finds. Let's see together :

Methods `insert` and `remove`

```
my_list = [1, 2, 3, 'picole']

## Add la picole
my_list.insert(1,'picole')
print(my_list)

## Remove la picole
my_list.remove('picole')
print(my_list)
```

```
[1, 'picole', 2, 3, 'picole']
[1, 2, 3, 'picole']
```

To concatenate two lists, there is the `extend` method which works as a `+=`.

Methods `extend` or `+=`

```
## Concatenate two lists
my_list = [1, 2, 3]
my_list2 = [4, 5, 6]
my_list.extend(my_list2)
print(my_list)

## OR

my_list = [1, 2, 3]
my_list += my_list2
print(my_list)
```

```
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

2.4.5 Browsing a list

Like for string, there is the keyword `in` for the list. You can browse through a list with a `for ... in ...` loop. The `in` will give the value of the browsing variable.

Browsing through a list 1/4

```
1 my_list = ['buuz', 'khushuur', 'khorkhog', 'spaghetti']
2
3 for element in my_list:
4     print(element)
```

```
buuz
khushuur
khorkhog
spaghetti
```

But I want the index Jeremy!
You could do something like this...

Browsing through a list 2/4

```
1 my_list = ['buuz', 'khushuur', 'khorkhog', 'spaghetti']
2
3 for i in range(0, len(my_list)): # function len() give
    the number of element of a list, a string, ...
4     print('i: {}, element: {}'.format(i, my_list[i]))
```

```
i: 0, element: buuz
i: 1, element: khushuur
i: 2, element: khorkhog
i: 3, element: spaghetti
```

It becomes a little messy, why not using a ready-to-go function that Python has: `enumerate`. When using this function, it returns a **tuple**. I'm not swearing. It's the correct term.

Browsing through a list 3/4

```
1 my_list = ['buuz', 'khushuur', 'khorkhog', 'spaghetti']
2
3 for element in enumerate(my_list):
4     print(element) ## (0, 'buuz') is a tuple !
```

```
(0, 'buuz')
(1, 'khushuur')
(2, 'khorkhog')
(3, 'spaghetti')
```

You can also separate the **tuple** using two different variables in the for loop like below.

Browsing through a list 4/4

```
1 my_list = ['buuz', 'khushuur', 'khorkhog', 'spaghetti']
2
3 for i, element in enumerate(my_list):
4     print('i: {}, element: {}'.format(i, element))
```

```
i: 0, element: buuz
i: 1, element: khushuur
i: 2, element: khorkhog
i: 3, element: spaghetti
```

A tuple works as a list BUT he cannot be modify after he's defined. If you try, it will return an error. I show it to you because you have to know it exists but you will almost never directly use it. Python will use it for you without even saying anything!

tuple

```
1 a = tuple((2, 'b')) # create a tuple
2 a[0] = 1
```

```
TypeError
<ipython-input> in <module>
----> 4 a[0] = 1
TypeError: 'tuple' object does not support item assignment
```

2.4.6 Exercises

For the Warmup 1 & 2 and the Exercises 1 & 2, we will use the list:

```
L = [1, 2, 3, 4]
```

Warmup 1

Assign the first element of a list to a variable.

Warmup 2

Insert an element to a specific index in a list.

Exercise 1

Write a script to sum all the items in a list using a for loop.

Exercise 2

Write a script to multiply all the numbers of a list with a constant.

Exercise 3

Write a script to get the biggest number from a list using a loop and a condition inside. For you knowledge, there is the function `max` who return you directly the maximum, but the goal here is to manipulate lists ...

Exercise 4

Write a script to copy a list. Be careful, you cannot just do `my_list1 = my_list2`.

Exercise 5

Write a script to append two lists together.

Exercise 6

Write a script to sum all the items in list given by a user. You have to use `input` but you cannot just ask once. You have to make a while-loop condition or a for-loop with a precise number of elements you want to put in ... and append in your list each time for you pass in the loop. Think about it.

Exercise 7

Write a script to get the last two elements of a list.

Exercise 8

Write a script to get a list with only the values of even indexes. $[15, 13, 59, 40]$ would give $[15, 59]$

Exercise 9

Write a script asking the user an index and then split the list into two different lists. $[1, 2, 3, 4]$ with the index 1 would give $[1, 2]$ and $[3, 4]$

Exercise 10

Write a script to insert a given string at the beginning of all items in a list. Example:

```
Input:
    L = [1,2,3,4]
    string = mongolia
Expected Output :
    ['mongolia1', 'mongolia2', 'mongolia3', 'mongolia4']
```

Exercise 11

Write a script displaying all the elements of the list that are less than 5.

```
a = [89, 1, 21, 3, 5, 8, 13, 2, 34, 55, 1]
```

Exercise 12

Write a script to create a 3X3 grid with numbers (Lists inside of a list).

```
Excepted output:
    [[1, 2, 3], [1, 2, 3], [1, 2, 3]]
```

Exercise 13

Write a script to sort a given list of numbers numerically (without the method `sort` of course).

Exercise 14

Let's say I give you a list saved in a variable: $a = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]$. Write one line of Python that takes this list a and makes a new list that has only the even elements of this list in it. There is something you can do: it's using a for loop in a list structure in one line ...

Exercise 15

Write a script that takes a list and print a new list that contains all the elements of the first list minus all the duplicates. (`if *** in ***`)

```
Input:
    a = [1, 4, 1, 16, 1, 36, 49, 64, 1, 100]
Excepted output:
    b = [1, 4, 16, 36, 49, 64, 100]
```

2.5. Dictionaries

A dictionary is the nearest object from a list. The difference between a list and dictionary is that a list contains other objects and you need the position to find it. A dictionary contains also objects but not in a particular order. They are sorted with **keys**.

For example, a dictionary can contain an address book and you access to the contact when you precise the name.

2.5.1 Create your first dictionary

To create your first dictionary:

Creation of a dictionary

```
1 my_dict = dict()
2 # or
3 my_dict = {}
```

Now you know that a list is defined with brackets [], a tuple with parentheses () and dictionary with braces {}.

To add a value, you just have the key you want to associate the value. Little example:

Adding values to the dictionary

```
1  # Creation of the dictionary
2  my_dict = {}
3
4  # Adding the value 91117788 to the key 'jeremow'
5  my_dict['jeremow'] = 91117788
6
7  # Adding the value 84708791 to the key 'zul'
8  my_dict['zul'] = 84708791
9
10 print(my_dict)
```

```
{'jeremow': 91117788, 'zul': 84708791}
```


If you want to change the value of a key, it's easy, just associate the key with a new value like this:

Modification of a value

```
my_dict['jeremow'] = 97778811
print(my_dict)
```

```
{'jeremow': 97778811, 'zul': 84708791}
```

If you call a key which doesn't exist, it will return a `KeyError`.

We used strings as keys, but you can also use numbers. It seems like a list with using numbers but the difference is, if you delete an index, it won't move all the others elements. Remember that a dictionary isn't ordered like a list.

Numbers used as keys

```
my_dict = {}

my_dict[0] = 'H'
my_dict[1] = 'e'
my_dict[2] = 'l'
my_dict[3] = 'l'
my_dict[4] = 'o'

print(my_dict)
```

```
{0: 'H', 1: 'e', 2: 'l', 3: 'l', 4: 'o'}
```

You can also fill a dictionary directly in one line:

Filling a dictionary in one line

```
my_dict = {0: 'H', 1: 'e', 2: 'l', 3: 'l', 4: 'o'}
print(my_dict)
```

```
{0: 'H', 1: 'e', 2: 'l', 3: 'l', 4: 'o'}
```

Imagine you want to play chess and remember where each piece are in a grid (from a to h, and 1 to 8):

Chessboard

```
chess_board = {}
chess_board['a', 1] = 'White Rook' # bottom left of the
    chess board
chess_board['b', 1] = 'White Knight' # next to the rook
chess_board['a', 2] = 'White Pawn' # in front of the rook
chess_board['b', 2] = 'White Pawn' # in front of the
    knight
#...
print(chess_board)
```

```
{('a', 1): 'White Rook', ('b', 1): 'White Knight', ('a',
   2): 'White Pawn', ('b', 2): 'White Pawn'}
```

What's happening? We can put several keys inside of a dictionary?
Not exactly, Python will interpret as ... a **tuple**! If you don't precise parenthesis here, you can see that the display is as a tuple. If you feared about forgetting that's a tuple, you can precise the parenthesis for Python but I personally think it's enough representative without!

2.5.2 Remove keys from dictionary

Something I didn't tell you before and it works with the variables is the keyword `del`. With it, you free the memory and delete the variable you don't want anymore.

Deleting a variable

```
a = 15
del a
print(a)
```

```
NameError
<ipython-input> in <module>
----> 3 print(a)

NameError: name 'a' is not defined
```

It works for everything but also for lists and dictionary to remove a specific element or key.

Deleting an element of a list

```
my_list = ['hey', 'you', 'shout', 'my', 'name']
del my_list[1]
print(my_list)
```

```
['hey', 'shout', 'my', 'name']
```

Deleting an element of a dictionary

```
my_dict = {'jeremy': 26, 'bayaraa': 'too old'}
del my_dict['jeremy']
print(my_dict)
```

```
{'bayaraa': 'too old'}
```

I'm not a big fan for lists and dictionaries to use it because methods exist to do it in a clean way. For lists, it was the `remove` method and for dictionaries, it's the `pop` method. You precise as parameter the key you want to delete.

Remove an element from a dictionary

```
my_dict = {0: 'H', 1: 'e', 2: 'l', 3: 'l', 4: 'o'}
print('original dict :\n', my_dict)

my_dict.pop(4)
print('Dict without the key 4:\n', my_dict)

my_dict.pop(1)
print('Dict without the key 1:\n', my_dict)
```

```
original dict :
 {0: 'H', 1: 'e', 2: 'l', 3: 'l', 4: 'o'}
Dict without the key 4:
 {0: 'H', 1: 'e', 2: 'l', 3: 'l'}
Dict without the key 1:
 {0: 'H', 2: 'l', 3: 'l'}
```

You know understand why it's not a list, when you remove an element, it doesn't change the number in `my_dict`... because it's a key, not an index.

Other interesting thing about the `pop` method: it returns the value you pop out of the dictionary. It can be useful to use it a last time before removing it.

Catch the removed element in a variable

```
1 my_dict = {'pseudo': 'jeremow', 'age': 26}
2
3 age = my_dict.pop('age')
4
5 print('Value removed:', age)
6 print('Updated dictionary:\n', my_dict)
```

```
Value removed: 26
Updated dictionary:
 {'pseudo': 'jeremow'}
```


2.5.3 Going through a dictionary

When you want to browse a dictionary, the `for`-loop will be useful again. You can get the keys, or the values or even both of them. It depends the method you use to browse the dictionary. The three methods will be `keys()`, `values()`, and `items()`.

Browsing a dictionary

```
1  fruits = {'watermelon': 15, 'apple': 10, 'peach': 25, '
       orange': 30}
2
3  print('Printing keys of fruits:')
4
5  for fruit_key in fruits.keys():
6      print(' ', fruit_key)
7
8  print('\nPrinting values of fruits:')
9
10 for fruit_value in fruits.values():
11     print(' ', fruit_value)
12
13 print('\nPrinting keys and values of fruits:')
14
15 for fruit_key, fruit_value in fruits.items():
16     print('  There are {} {}'.format(fruit_value,
           fruit_key))
```

```
Printing keys of fruits:
  watermelon
  apple
  peach
  orange

Printing values of fruits:
  15
  10
  25
  30

Printing keys and values of fruits:
  There are 15 watermelons
  There are 10 apples
  There are 25 peachs
  There are 30 oranges
```

2.5.4 Exercises

Warmup 1

Remove the element of the key `key4`.

```
my_dict = {'key1': 'hello', 'key2': 13, 'key3': [1, 2,
    3], 'key4': True}
# ...
print()
```

Warmup 2

Add the boolean `False` to the key `key4`.

```
my_dict = {'key1': 'hello', 'key2': 13, 'key3': [1, 2,
    3]}
# ...
```

Warmup 3

Loop into the dictionary `fruits` and print the keys of it.

```
fruits = {'watermelon': 15, 'apple': 10, 'peach': 25, '
    orange': 30}
for # ...
```

Warmup 4

Loop into the dictionary `fruits` and print the values of it.

```
fruits = {'watermelon': 15, 'apple': 10, 'peach': 25, '
    orange': 30}
for # ...
```

Warmup 5

Loop into the dictionary `fruits` and print the keys and values of it.

```
fruits = {'watermelon': 15, 'apple': 10, 'peach': 25, '
    orange': 30}
for # ...
```

Exercise 1

Filter a dictionary based on values: you have a dictionary with name as keys and ages, and you want to create another dictionary with only the people under 18.

```
people = {'Jeremy': 26, 'Zayaa': 4, 'Zul': 16, 'Khulan':
    33, 'Tuguldur': 12, 'Buyanaa': 48}
children = {}
# ...
```

Exercise 2

Find the highest value in the dictionary.

```
people = {'Jeremy': 26, 'Zayaa': 4, 'Zul': 16, 'Khulan':
    33, 'Tuguldur': 12, 'Buyanaa': 48}
# ...
```

Exercise 3

Get the highest price in a shop and put it in tuple.

```
shop = {'apple': 6, 'pear': 12, 'coconut': 42, 'mango':
    33, 'ananas': 15}
# ...
```

Exercise 4

Filter a dictionary based on values. For example for values over 30:

```
shop = {'apple': 6, 'pear': 12, 'coconut': 42, 'mango':
    33, 'ananas': 15}
# ...
```

```
{'coconut': 42, 'mango': 33}
```

Exercise 5

Write a script to create a dictionary from a string. Be careful: You have to count when a letter appears several times.

```
my_string = 'erdenezul'
# ...
```

```
{'e': 3, 'r': 1, 'd': 1, 'n': 1, 'z': 1, 'u': 1, 'l': 1}
```

Exercise 6

Date Decoder. A date of the form 8-MAR-2021 includes the name of the month, which must be translated to a number. Create a dictionary suitable for decoding month names to numbers. Use string operations to split the date into 3 items using the "-" character.

It will accept a date in the `"dd-MMM-yyyy"` format and respond with a list of `[y, m, d]`.

```
Input:
    "8-MAR-2021"
Output:
    [2021, 3, 8]
```

Exercise 7

You have a list of students in a list. It prints `True` if there is the value of the specified key exist. Else `False`.

```
students = [
        {'student_id': 1, 'name': 'Jeremy', 'class': '1'
           },
        {'student_id': 2, 'name': 'Tuguldur', 'class': '3
           '},
        {'student_id': 3, 'name': 'Pujee', 'class': '2'},
        {'student_id': 4, 'name': 'Zul', 'class': '2'},
        {'student_id': 5, 'name': 'Demberel', 'class': '1
           '}
        ]

key = 'name' # you can change to test
value = 'Demberel' # you can change to test -> it has to
   return true

#...
```

Exercise 8

For this exercise, we will keep track of when our friend's birthdays are, and be able to find that information based on their name. Create a dictionary (in your file) of names and birthdays. When you run your script it should ask the user to enter a name, and return the birthday of that person. The interaction should look something like this:

```
>>> Welcome to the birthday dictionary. We know the
   birthdays of:
Albert Einstein
Benjamin Franklin
Ada Lovelace
>>> Who's birthday do you want to look up?
Benjamin Franklin
>>> Benjamin Franklin's birthday is 01/17/1706.
```

(Albert Einstein: 03/14/1879 ; Benjamin Franklin: 01/17/1706 ; Ada Lovelace: 12/10/1815)

3. Functions & Modules

In this part, we'll talk again about function. It is really one of the most important thing to understand perfectly. Without this, you cannot go further in Python. We will go slowly, don't panic!

When programming, we must often use some part of the code several times. Not just with loop but grouped within bigger code. This can be function or module. I'll go easy on these two concepts but detailed enough for you to understand all what you need to follow the lead of programming.

Functions allow you to group several commands/instructions with a name. You already saw the function `print` for example.

Modules allow you to group several *Functions* with the same principle. For example, all the mathematical functions are in module named `math`.

And at least, we will apprehend the **exceptions** in Python. A way to **traceback** the errors we can make.

3.1. Functions

3.1.1 Create a function

First, a function is defined by different rules:

Create a function

```
1 def name_function(parameter1, parameter2, parameter3,
      parameterN):
2     # some code here
```

1. `def` is the keyword for define. Python know there will be a function after this word
2. The name of the function which will be used as a variable after. Don't name a function after something which is already used like `print` or `type`
3. The list of parameters with a comma between then and a space for the visibility. When no argument is passed, the brackets stays compulsory
4. Two points at the end.

Function without parameters

```
1 def name_function():
2     # some code here
```


We are now in a karaoke bar where Baby Girl song can be sung. The waiter is here and wants our first round of drinks. Jeremy, Bayaraa, Pujee, Munkhbayar and Tuguldur take a beer. The function to compute the sum of what we'll pay could be:

Function `first_order`

```
1 def first_order():
2     # define the order and the price
3     nb_beer = 5
4     price_beer = 4000
5
6     total_price = nb_beer * price_beer # compute the
          price
7     print('The total price for 5 beers is :',
8           total_price)
```

Nothing happens when this script is launched because we didn't call the function for the moment.

Watch to the names of the variables. They are really precise and we know exactly what they will contain. It is absolutely necessary to put names like this. Not `b = 5` and `p = 4000` for example. When you'll watch to your code again in three months, if you see `p` and `b`, you won't understand anymore what was the meaning of all of this. Same with the comments. If you explain a little what you're doing, you will remember three months later why you did this and also some other people who read your code can also understand what's going on.

To call the function and use it, you have to the line below.

Calling function `first_order`

```
1 first_order()
```

Euh, za za, sain baina but what the point of this, was it necessary to create a function for this? Not really but what if we want to take another order and Pujee is already gone because we have to do the maintenance in Songino on a saturday morning. It will be more useful. The `nb_beer` variable will now be a parameter. The code will be easy to find:

Function order

```
1 def order(nb_beer):
2     price_beer = 4000 # define the price
3     total_price = nb_beer * price_beer # compute the
          price
4     print('The total price for {} beers is: {}'.format(
          nb_beer, total_price))
5
6 # launch the function
7 order(5)
```

```
The total price for 5 beers is: 20000
```

Mmh, what's happening with the `print` function? I didn't see that before.
Yes there are different ways of printing variables. The first you know is:

print format 1/2

```
1 a, b = 1, 2
2 print('a =', a, ', b =', b)
```

```
a = 1 , b = 2
```

The new way is:

print format 2/2

```
1 a, b = 1, 2
2 print('a = {}, b = {}'.format(a, b))
```

```
a = 1 , b = 2
```

You're seeing that it displays the same thing. With the help of `.format()` (a method from the class `string`) you can add the variables at the end of the `print` function which is a little more readable as the first solution. However, both options are correct. I just wanted to show you another option of printing some text.

Back on topic, we can add more parameters if we want like this:

`order` function with two parameters 1/3

```
1 def order(nb_beer, price_beer):
2
3     total_price = nb_beer * price_beer # compute the
          price
4     print('The total price for {} beers is: {}'.format(
          nb_beer, total_price))
```

Now, if you call the function, you can choose the number of beers your order and the price of one beer.

`order` function with two parameters 2/3

```
1 order(5, 5000)
2 order(4, 3000)
```

```
The total price for 5 beers is: 25000
The total price for 4 beers is: 12000
```

If you just write the parameters without naming it, you have to respect which parameter is number 1, number 2 ... but you can mix them if you named it explicitly when the function is called.

`order` function with two parameters 3/3

```
1 order(nb_beer = 5, price_beer = 5000)
2 order(price_beer = 5000, nb_beer = 5)
```

```
The total price for 5 beers is: 25000
The total price for 5 beers is: 25000
```

3.1.2 Default value of a parameter

You can tell a function what is the default parameter if the user doesn't precise it and it's really easy.

`order` function with default parameter 1/2

```
def order(nb_beer, price_beer=4000):
    """
    Function which print the price of a specific order of
        beers

    (nb_beer >= 0)
    """

    total_price = nb_beer * price_beer # compute the
       price
    print('The total price for {} beers is: {}'.format(
       nb_beer, total_price))
```

You just need to add `=4000` for the parameter `price_beer`. Easy, right?

`order` function with default parameter 2/2

```
order(3)
order(3, price_beer=5000)
```

```
The total price for 3 beers is: 12000
The total price for 3 beers is: 15000
```

I also add some text under the definition of the function between `""" """`. It is called `docstring`. It will help the user or another people who wants to use your code to see what the function does. To get the help you just have to write `help(order)`

`docstring` of the function `order`

```
help(order)
```

```
Help on function order in module __main__:

order(nb_beer, price_beer=4000)
    Function which print the price of a specific order of
       beers

    (nb_beer >= 0)
```

3.1.3 Signature of a function

In Python, you cannot precise the type of the parameters of the function like in other languages like Java, C++. The signature of the function will be the name itself. It means that if you write a new definition of the function, it erases the previous one even if the parameters aren't the same inside.

Signature of a function

```
1 def say_hello(name):
2     print('Hello,', name)
3
4 say_hello('Jeremy')
5
6 def say_hello():
7     print('Sain uu?')
8
9 say_hello('Jeremy')
```

```
Hello, Jeremy

---------
TypeError
<ipython-input> in <module>
----> 9 say_hello('Jeremy')

TypeError: say_hello() takes 0 positional arguments but 1
   was given
```

In other languages, we call this situation **overload** of a function. That's not possible in Python: **One name, one function**.

3.1.4 The instruction `return`

`return` in a function is the way to get a value from it. In the previous examples, we just printed to values on the screen but you cannot reuse it later. Perhaps you will take several orders during the night and with the effect of alcohol, you don't remember what was the price of everything. Hopefully the barman was adding each time on his computer what you ordered. Now we don't print anymore the `total_price` but the function will return it and you can print it if you want. We even change the name as the function doesn't give us the same information. A name nearer from the reality of the function is now `order_price`.

`return` in a function 1/2

```
1  def order_price(nb_beer, price_beer=4000):
2      """
3      Function which returns the price of a specific order
           of beers
4
5      (nb_beer >= 0)
6      """
7
8      return nb_beer * price_beer
9
10 total_price = order_price(nb_beer=3)
```

Why does it do nothing?
It computes the total price but you just didn't print it! Add a `print(total_price)` after the line 10 and you'll get the result of the function displayed.

You can also return several values if you want. They just have to be separated by a comma. For example, imagine you ordered also some Airag and you want the details of the invoice:

`return` in a function 2/2

```
def order_price(nb_beer, nb_airag, price_beer=4000,
    price_airag=6000):
    """
    Function which returns the total price of a specific
        order.

    (nb_beer >= 0, nb_airag >= 0)
    """
    price_beer = nb_beer * price_beer
    price_airag = nb_airag * price_airag

    return price_beer, price_airag, price_beer +
        price_airag

price_beer, price_airag, total_price = order_price(3, 4)
print('Beers : {}T\nAirag : {}T\n-------\nTotal : {}T'.
    format(price_beer, price_airag, total_price))
```

```
Beers : 12000T
Airag : 24000T
-------
Total : 36000T
```

Watch to the format of the string, you already did it before... `\n` is the escape character to create a new line.

3.1.5 Function with unknown number of parameters

Perhaps you will need to declare some functions without knowing at first how many parameters in input you will use. The easiest way to declare it is like this : `def function(*parameters)`. We simply write a little star in front of the argument which will receive the list... or I must say the tuple (you see it before, you know it). A little example with a really strange function that nobody can understand : the `add` function.

Function add with unknown number of parameters

```
def add(*numbers):
    """
    Function + to add numbers between them.
    """

    sum_nb = 0 # initialization

    for i, number in enumerate(numbers): # no star when
        you call the parameter after !
         sum_nb += number

    return sum_nb

res = add(1,2)
print('1+2={}'.format(res))

res = add(5,5,10,10)
print('5+5+10+10={}'.format(res))
```

```
1+2=3
5+5+10+10=30
```

It sees useless to create an `add` function for numbers when you can just use `+` and you're right. We will also see in chapter four **Object-Oriented Programming** how to change the behavior of the `+` sign when you will create your own classes... you already saw some different behaviors depending on the classes : `string`, `integer`, `complex` ...

Different behavior of the + sign

```
print('Hello ' +  'World!') # str
print(3 + 4) # int
print((3+4j) + (1+2j)) # complex
```

```
Hello World!
7
(4+6j)
```

You can declare compulsory parameters before a tuple of parameters like `def function(name, age, *parameters)`. Caution! you cannot declare first a list of param-

eters. If you write this: `def function(*parameters, name, age)`. You understand that if you don't name the parameters when you call to the function, Python cannot know where your optional parameters end...

The exception is when you declare a function with predefined parameters like `def function(name, age, *parameters, color='blue', place='Mongolia')`. It's made like this because when you want to change the predefined value of the parameter, you have to call it by its name. So Python cannot misinterpret your list of **unknown parameters** and the **predefined parameters**.

And it's now the rule when you write a function, always declare in this order when you need all of the three types :

1. Compulsory parameters
2. Tuple of parameters
3. Predefined parameters

3.1.6 To go further: `lambda` function

We just saw how to write a function with the `def` keyword. Python has another feature which is called **`lambda` function**. They are limited for just one instruction. It's a really short function. I won't argue for the moment why it exists. You just cannot say you don't know.

The syntax is quite simple:

`lambda` syntax

```
lambda arg1, arg2, ... : instruction
```

For example, for the order of beers:

`lambda` function applied to `order_price` 1/3

```
lambda nb_beer, price_beer : nb_beer * price_beer
```

```
<function __main__.<lambda>(nb_beer, price_beer)>
```

The output is strange, why have I this?
You just declare the lambda without saving it somewhere. You have to store it in a variable to use it later. You can declare it like this and you can use it like any function.

`lambda` function applied to `order_price` 2/3

```
1 order_price = lambda nb_beer, price_beer : nb_beer *
    price_beer
2
3 order_price(2, 4000)
4 order_price(price_beer = 6000, nb_beer = 3)
```

```
8000
18000
```

You can even add a default value.

`lambda` function applied to `order_price` 3/3

```
1 order_price = lambda nb_beer, price_beer=5000 : nb_beer *
    price_beer
2
3 order_price(3)
```

```
15000
```

It seems useless for the moment, and you're right. Our level in Python doesn't need this definition. But we're in the good section to have the first feeling about it. And when you'll see it again, you'll ask me **What is this lambda thing, lord Jérémy?** and I'll answer **See chapter three guys**.

3.1.7 Exercises

For the Warmups and the Exercises, I will write some code which is only the **test** part. You have to write your script before and adding the code I give at the end, it has to give you the correct output.

Warmup 1

Create a function `say_hello` which displays the message *"ooooo sain baina uu?"*

```
say_hello()
```

Warmup 2

Create a function `say_hello` which takes one parameter `name` and display *"`name`, sain baina uu?"*

```
say_hello('Baigalaa')
```

Warmup 3

Create a function `present_user` which takes 2 parameters `name` and `age` and displays *"Minii ner `name`. Bi `age`-n nastai"*

```
present_user('Jeremy', 26)
```

Warmup 4

Create a function `say_hello` which takes one parameter `name` and display *"`name`, sain baina uu?"*. Default value of `name` if not precised is *Zochin*.

```
say_hello('Tunga')
say_hello()
```

Warmup 5

Create a function `get_2x_age` which takes `age` as parameter, displays *"You're* `age` *and twice your age is :* `age_2x`*"* and return two times your age. Default value is *18*.

```
1 twice_my_age = get_2x_age(26)
2 print(twice_my_age)
3
4 twice_my_age = get_2x_age()
5 print(twice_my_age)
```

Exercise 1

Create a function `above_limit` which takes in parameter a list `L` and an integer `limit` and **return** a new list with only the numbers of `L` which are above or equal to `limit`.

```
1 my_list = [2, 15, 18, 25, 96]
2 above_list = above_limit(my_list, 16)
3 print(above_list) # must display [18, 25, 96]
```

Exercise 2

Create a function `first_elements` which takes a list `L` and an integer `nb_elements` in parameters and **return** a new list with only the first `nb_elements` elements of the list `L`. If `nb_elements` is superior to the length of the list `L`, return a new list with all the elements of `L` (don't return `L` itself!). To get the length of a list: `len(my_list)`.

```
1 my_list = [2, 15, 18, 25, 96]
2 my_new_list = first_elements(my_list, 2)
3 print(my_new_list) # must display [2, 15]
4
5 my_new_list = first_elements(my_list, 10)
6 print(my_new_list) # must display [2, 15, 18, 25, 96]
```

Exercise 3

Create a function `last_elements` which takes a list `L` and an integer `nb_elements` in parameters and **return** a new list with only the last `nb_elements` elements of the list `L` beginning from the end. If `nb_elements` is superior to the length of the list `L`, return an empty list. To get the length of a list: `len(my_list)`.

```
my_list = [2, 15, 18, 25, 96]
my_new_list = last_elements(my_list, 2)
print(my_new_list) # must display [96, 25]

my_new_list = last_elements(my_list, 10)
print(my_new_list) # must display []
```

Exercise 4

Create a function `fact` which takes a number `n` as parameter and **return** the factorial of this number. It must be a non-negative integer. If `n` negative, display *"Number must be positive"* and return `None`. (Be careful, `fact(0) = 1` ...)

```
nb = fact(5)
print(nb) # must display 120

nb = fact(-1) # must display "Number must be positive"
print(nb) # must display None
```

Exercise 5

Create a function `case_letters` that takes a string `string` as parameter and **displays** the number of upper case letters and lower case letters. The method to see if a character is in lower case: `char.islower()` return `True` if `char` is in lower case. For upper case: `char.isupper()` ...

```
case_letters('JeReMY') # must display: "Number of lower
    case letters: 2 ; Number of upper case letters: 4."
case_letters('Hello, I am a real Mongolian Woman')
# must display: "Number of lower case letters: 23 ;
    Number of upper case letters: 4."
```

Exercise 6

Create a function `is_pangram` which takes a string as parameter and **displays** whether the string is a pangram or not and **return** `True` or `False`. A pangram is a sentence with every letter of the alphabet at least once. To do it, you have to check every letter of the sentence and add to a defined dictionary the number of times you see a letter. I give you the dictionary you can copy:

```
alphabet = {'a': 0, 'b': 0, 'c': 0, 'd': 0, 'e': 0, 'f':
   0, 'g': 0, 'h': 0, 'i': 0, 'j': 0, 'k': 0, 'l': 0, 'm':
    0, 'n': 0, 'o': 0, 'p': 0, 'q': 0, 'r': 0, 's': 0, 't
   ': 0, 'u': 0, 'v': 0, 'w': 0, 'x': 0, 'y': 0, 'z': 0}
```

If one of the value associated to the key is equal to 0, that means that's not a pangram. The french sentence *"Portez ce vieux whisky au juge blond qui fume"* contains exactly each letter only once for example.

```
1 answer = is_pangram('Portez ce vieux whisky au juge blond
      qui fume') # must display: "The sentence is a pangram
     "
2 print(answer) # must display True
3
4 answer = is_pangram('Hello, I am a real Mongolian Woman')
      # must display: "The sentence isn't a pangram"
5 print(answer) # must display False
```

Exercise 7

Write a little game **More Or Less** as a function `more_or_less` that randomly generates a number between two given numbers and ask the user to give a number until he finds it. You will write the function `more_or_less` which takes two parameters: the limits of the game. You'll also write the function `compare_number` which takes two numbers as parameters (the one which is generated by the game and the input one of the user) and displays if the number to guess is higher or lower than the user number and return `False`, displays *"You win"* if you find the number and return `True`.

To generate randomly, we will use a function called `random.randint(a, b)` with `a` and `b` the range where the number will be chosen randomly from a library called `random` (we'll see next section what it is).

```
1  import random
2
3  def compare_numbers(nb_to_guess, user_nb):
4      if #...
5          print()
6          return
7      elif #...
8          print()
9          return
10     else:
11         print()
12         return
13
14 def more_or_less(low_lim, high_lim):
15
16     nb_to_guess = random.randint() # complete here
17     user_win = False
18
19     while user_win is not True:
20         # input number and convert into int ...
21         user_win = compare_numbers() # complete
22
23 # here's the game to play
24
25 # to test the game at first, it will be easier if the
       range is really tight.
26 # When it works properly, you can change the high limit
       to 50 for example.
27 more_or_less(1, 4)
```

Exercise 8

The `print` function: Try below to rewrite the print function. Some indications : the function doesn't return anything, you have to convert all the arguments in the string format before sending your final string to the stream. The stream is where you display the printed object you want to see or users to see. It's a little bit soon before Modules and Packages is the next lesson but just assume this:

```
1 import sys
2 sys.stdout.write('Hello World!\n')
```

```
Hello World!
```

It's the equivalent of this:

```
1 print('Hello','World!')
```

```
Hello World!
```

All the input parameters have to be automatically separated with a space and the final string has to end with the backspace character `\n`. Now you have all the keys in your hand, go and fly by yourself!

```
1  import sys
2
3  def your_print(*parameters, sep = ' ', end = '\n'):
4      """
5      Here is your print function. The parameters are in a
          tuple, the sep character is a space and the ending
6      character is backspace.
7      """
8
9      # get all the parameters one by one, convert them to
          str and add them the a final string called
          final_string
10
11
12     # here the last line to write on the stream :
13     sys.stdout.write(final_string)
14
15 your_print('Hello','World!') # must print 'Hello World!'
16 your_print('Hello you')
```

Exercise 9

Open the software you use (PyCharm or Spyder) and create a new Python file called `shop.py`. If you don't have one, you can work here directly.

You manage a fruit shop. Your wholesaler has a list of fruits available in infinite amount you can buy. His list will be composed of orange, apple, grape, watermelon, peach, banana, apricots, cherries. You don't have to buy all of the fruits. You store them in a dictionary. If you don't have the product, it disappears from your dictionary. A client can come in your shop and buy fruits if there are enough.
You'll create four functions:

1. `buy_fruit_from_wholesaler`
2. `sell_fruit_to_client`
3. `print_wholesaler`
4. `print_storage`

Don't be afraid, I don't let you like this. Here's some code to begin you can copy:

Function `print_storage`

```
1  # -*- coding: utf-8 -*-
2
3  def print_storage(storage_dict):
4      """
5      Function to display the storage of our fruits shop
6      If the storage_dict is empty, print: 'Sorry, no
           article available'. Else print the details of the
           shop.
7
8      Return nothing
9      """
10
11     print('Welcome to my supermarket! Here\'s what we
           have today:')
12
13     if :
14     else:
15         for :
```

Function `print_wholesaler`

```
1  def print_wholesaler(wholesaler_list):
2      """
3      Function to display the available fruits in the
           wholesaler
4
5      Return nothing
6      """
7
8      print('Welcome to the wholesaler! Here\'s the fruits
           available in infinite quantity:')
9
10     for :
```

Function `sell_fruit_to_client`

- Verify if there is enough fruits in the `storage_dict`:
  - If there is enough, it substracts the number from the dictionary.
  - If there isn't enough, it says to the client how many there are left.
  - If the article doesn't exist, it says Sorry, we don't have this article.
- If an article has an amount of 0, it will pop out of the dictionary.
- No need to return anything, the dictionary is a reference ... That means that when the job with the function is done, it automatically updates the dict without returning it. (I put an example below to get the concept of reference: see **Little explanation of difference between value and reference**)

Function `sell_fruit_to_client`

```
1  def sell_fruit_to_client(storage_dict, fruit, quantity):
2      """
3      Function to sell to a client a fruit in a certain
           amount quantity.
4      """
5
6      if :
7      else:
8          if :
9          elif :
10         else:
```

Function `buy_fruit_from_wholesaler`

Verify if the article exist in the `wholesaler_list`, if you have already the article you want yo buy:

- If no, it adds it to your dictionary.
- If yes, it adds to the amount already written in it.

Function `buy_fruit_from_wholesaler`

```
1  def buy_fruit_from_wholesaler(fruits_list, storage_dict,
       fruit, quantity):
2      """
3      Function to buy a certain fruit in the quantity you
           want and add it to you storage_dict.
4      """
5
6      if :
7      else:
8          if :
9          elif :
10         else:
```


Main script to test the functions

```
### Here we test the functions ...
if __name__=='__main__':

    # our shop is here :
    fruits_shop = {}

    # The wholesaler list is here
    wholesaler_list = ['orange', 'apple', 'grape', '
       watermelon', 'peach', 'banana', 'apricot', 'cherry
       ']

    print_storage(fruits_shop) # Nothing for the moment

    buy_fruit_from_wholesaler(wholesaler_list,
       fruits_shop, 'banana', 25)
    buy_fruit_from_wholesaler(wholesaler_list,
       fruits_shop, 'apple', 17)
    buy_fruit_from_wholesaler(wholesaler_list,
       fruits_shop, 'watermelon', 91)
    buy_fruit_from_wholesaler(wholesaler_list,
       fruits_shop, 'avocado', 10) # fruit isn't in the
       wholesaler list

    sell_fruit_to_client(fruits_shop, 'banana', 5)
    sell_fruit_to_client(fruits_shop, 'apple', 17)
    sell_fruit_to_client(fruits_shop, 'cherry', 40) #
       fruit isn't in the storage
    sell_fruit_to_client(fruits_shop, 'watermelon', 92) #
        too much, he cannot buy it

    print_storage(fruits_shop) # you normally get 20
       bananas and 91 watermelons.
```

Exercise 9 - Advanced

Make modifications on your code on the following functions:

- `sell_fruit_to_client(storage_dict)`
- `buy_fruit_from_wholesaler(fruits_list, storage_dict)`
- Delete the parameters of `fruits` and `quantity`
- Ask the user to write it when launching the program (the function `input()`...)

Little explanation of difference between value and reference

What's happening with a as a number? (or a string)

```
1  a = 15
2  print('Value of a :', a)
3
4  def change(a,b):
5      a = b
6      print(a)
7
8  print('value of a in the function :')
9  change(a, 5)
10
11 # The value isn't change out of the function
12 print('Reprint a :', a)
```

```
Value of a : 15
value of a in the function :
5
Reprint a : 15
```

What's happening with a as a dictionary? (or a list)

```
1  a = {'number': 15}
2  print('Value of a :', a)
3
4  def change(a, b):
5      a['number'] = b
6      print(a)
7
8  print('value of a in the function :')
9  change(a, 5)
10
11 # The value is changed even out of the function
12 print('Reprint a :', a)
```

```
Value of a : {'number': 15}
value of a in the function :
{'number': 5}
Reprint a : {'number': 5}
```

3.2. Modules & Packages

Modules allow you to group several **functions** with the same principle. For example, all the mathematical functions are in module named `math`.

We just used functions already loaded in Python for the moment but the number of functions outside already written allow us to do a lot more.

A module is simply a file with functions and variables. If you want to use it, you have to import it in your program. They cannot be directly imported in your code because they have to be interpreted first.

3.2.1 The method `import`

When you want to import a module, you just need to call the name of the library[1] you want like `math` and use the keyword `import`.

Import a library

```
1 import math
```

Now you imported the module. You can simply use all the functions inside of it. You just have to call the function with the `module_name` + `'.'` + `function_name`. For example: `math.sqrt` is the square root function.

Function `sqrt` of the module `math`

```
1 math.sqrt(25)
```

```
5.0
```

[1] That's not totally accurate because if you don't have it on your computer, you need to download it (see https://docs.python.org/3/installing/).

If you want to know what functions are inside of a module, you can write `help("math")` in the interpreter and then move with Enter to go on line by line ; *Space* to go one page further and *Q* to quit the help (not working like this in Jupyter platform but in interpreter on your computer yes). However, I advise you to directly go on the documentation on the internet for each module. It's really well designed and you can search in a better way for what you need. (For `math` module : https://docs.python.org/3/library/math.html)

3.2.2 Changing the name of a module

You can also change the name of the module. So for `math`, the example is not really appropriated. But you can change it anyway. The keyword to use is `as`.

Create an alias of a module

```
1 import math as m
2 m.sqrt(25)
```

```
5.0
```

It is called a workspace. Now your workspace of `math` module is called `m`. We'll see later but it's useful for some other module like `matplotlib`, the graphical library of Python. I don't explain so much but for example you can write `import matplotlib.pyplot as plt`. You see now why it can be useful, you time is precious.

3.2.3 `from ... import ...`

You can also just import some functions of a module without getting all of them. This method allow you to get some functions and after you don't need the name of the module anymore.

`from ... import ...`

```
1 from math import sqrt
2 sqrt(25)
```

```
5.0
```

Now you have the `sqrt` function as if it was in your script code. You can import several functions with a comma between them: `from math import sqrt, cos`.

Other great thing to know but you have to be careful about it. You can import all the functions of a module without the name of it with `from ... import *`. For example with the module `math`:

from ... import *

```
1 from math import *
2 acos(pi/4) # arcosine function and pi constant
```

```
0.6674572160283838
```

You have to pay attention to it if the same function is in several modules. It can interfere between them and you don't know where the function comes from. In this case, to have no doubt, you import the module with his name. You really know the function is from `math` module for example and not other imported modules.

3.2.4 One file to control them all

As you already know, you can write instructions in the interpreter but also create file directly in your software. The two most common are *PyCharm* and *Spyder*. *PyCharm* stays at the top-professional level of all software. *Spyder* is more educational. Now the real work begins.

You will open your software and create a new file. You copy/paste this code in it.

Your first file

```
1  # -*- coding: utf-8 -*-
2
3  import os # this library is useful to communicate the
       operating system of your computer
4
5  # we define our function to get the order price :
6  def order_price(nb_beer, price_beer=4000):
7      """
8      Function which returns the price of a specific order
           of beers
9
10     (nb_beer >= 0)
11     """
12     return nb_beer * price_beer
13
14 nb_beer = input("How many beers do you want to drink ? ")
       # asking the user to give his order
15 nb_beer = int(nb_beer) # convert
16 total_price = order_price(nb_beer) # compute the price
17 print('Your order of {} beer(s) will cost you {}T'.format
       (nb_beer, total_price))
18
19 os.system('pause') # this line is used by Windows to
       avoid quitting the terminal when you launch it
```

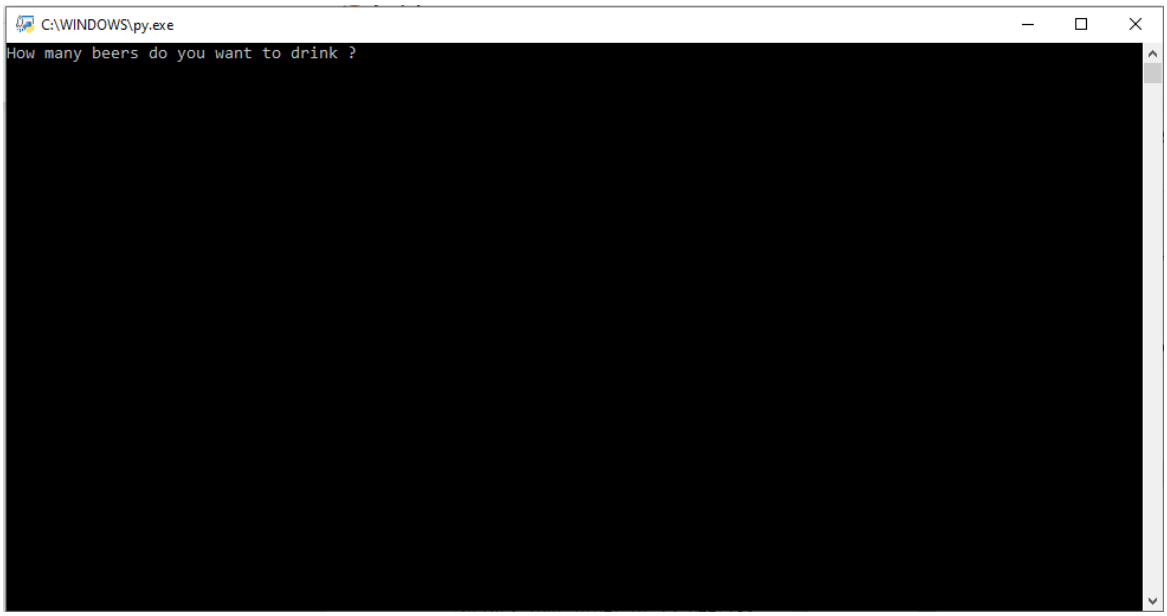
The point is to practise on your own computer but you can use the resources of Jupyter notebook if you want[2].

What's this first line?
We need it on your computer if you use accents. Perhaps not so many in english but for example in my french name Jérémy. That's not a lesson about encoding but utf-8 will be universal for languages with latin letters.

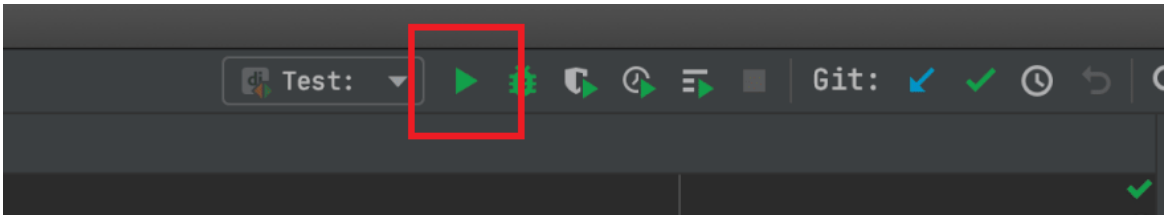
[2] https://github.com/jeremow/Python_lessons

If you run on Linux you have to add this line before all, it's the path where the python interpreter is: `#!/usr/bin/python3.8`. It can change according to the version of Python you have and where you installed it).

If you run the file you just saved by double clicking on it, you'll see this on Windows:



You can also launch it directly in PyCharm using this:



Okay Jérémy, but where do we go now? What are we doing?
Yeah sorry, that was really important to understand that before following the lesson about modules. If you knew about all of this, it doesn't kill after all this time to see it again!

3.2.5 Conquer the world and create your own modules

Time to create your module. Create a folder on your computer and with PyCharm, create two files:

- `bar.py` which will contain our function `order_price`
- `test.py` which execute some test on your module

The code of `bar.py`:

bar.py

```
1  # -*- coding: utf-8 -*-
2
3  """module bar with the function order_price"""
4
5  def order_price(nb_beer, price_beer=4000):
6      """
7      Function which returns the price of a specific order
           of beers
8
9      (nb_beer >= 0)
10     """
11     return nb_beer * price_beer
```

It's for the first example, it seems useless for the small amount of things we wrote but you have to understand how it works when you will be surrounded by thousands lines of code. I added some `docstring` to comment the module. Keep the habit to explain what the function, files are for.

And here for the `test.py` file:

test.py

```
1  # -*- coding: utf-8 -*-
2
3  import os
4  from bar import * # or import bar
5
6  nb_beer = input("How many beers do you want to drink ? ")
7  nb_beer = int(nb_beer)
8
9  # test of the function order_price
10
11 total_price = order_price(nb_beer) # or bar.order_price(
       nb_beer)
12 print('Your order of {} beer(s) will cost you {}T'.format
       (nb_beer, total_price))
13
14 os.system('pause')
```

Et voilà! Good to know it was that simple. Now you can create everything you want and further ... and it will be the next step after studying the exceptions.

You will see when running `test.py` that a folder `__pycache__` has been created. It's

a compiled code used by Python to run faster the next time you'll run `test.py`. Don't be afraid, you don't have to be preoccupied by this, it's a story between Python and the OS.

3.2.6 Make a test directly in the module

It would be cool if we can test directly in the module file without affecting the other files when importing our module. And it's possible. But if you just write after your `order_price` function: `order_price(3)` for example, when importing your module, this line will also be executed. That means that you'll see everytime the result for three beers even if you don't want to call the function. To avoid this, you'll create a new condition in your module:

Having test lines in your module

```
1  # -*- coding: utf-8 -*-
2
3  """module bar with the function order_price"""
4
5  import os
6
7  def order_price(nb_beer, price_beer=4000):
8      """
9      Function which returns the price of a specific order
           of beers (nb_beer >= 0)
10     """
11     return nb_beer * price_beer
12
13 # test of the function order_price
14
15 if __name__ == "__main__":
16     total_price = order_price(3)
17     print('Price:', total_price)
18     os.system('pause')
```

```
Price: 12000
```

When you launch your module `bar.py`, it executes the function for three beers. However, if you launch `test.py`, it doesn't run this part of the code. What magic trick is this?

With the variable `__name__`, the interpreter Python know if you run the file as the main file or if you import it in another file like `test.py`. So if you run `bar.py`, Python considers it as the main file (obviously) and so it will run the code inside of it.

3.2.7 Packages

A **module** encapsulates several **functions** and other things (chapter **Object-Oriented Programming**, wait for it, you're not ready). And a bit higher is what we called a **package**.

Theory

The theory behind this is the organization. When you create a big software, you'll separate the things which don't match together. It's good to know what you need and what you don't need in every element. Imagine with our `bar` module if we write functions about the drinks like before but also about the workers inside it and why not the clients. Absolutely, you won't put all your functions and variables in just one file. It will be a mess and don't find what you need. Even worse for people who doesn't know how your module works.

So we separate things in packages like `workers`, `clients` and `drinks`. And the final goal is to create a **library** with packages inside of it. It's the best for the hierarchy. And why? Because you won't have to know all the packages, modules and functions to use your library but just what you need.

Practice

It is just ... folders with files inside! For our example of the bar, the hierarchy will be:

- The name of the library, a folder with the name `bar`
  - a folder `people` with
    - a module `client.py`
    - a module `worker.py`
  - a module `order.py`
  - ...

To import some packages in your program, it always with the keyword `import` and `from`.

Import the module

```
1 import bar
```

With this, you import all the library and then you can call some modules inside of it with a point between names.

Call subpackage and module

```
bar.people # call the subpackage people
bar.people.client # call the module client
```

If you just need one module or even just one function inside you can use the `from ... import ...` syntax.

Call only a part of the package

```
from bar.people import client # import the module
from bar.people.client import get_name # import just one
    function of the client module
```

One last example, if you put in `order.py` our function to get the price of the order. When you want to import it from our library and use it, you have two possibilities.

Two ways to call a function

```
from bar.order import order_price
order_price(3) # call the function order_price

### OR ###

import bar.order
bar.order.order_price(3) # call the function order_price
```

Now you understand everything, we'll see how to manage exceptions and a big exercise! I'm pride of you all, you read it till there and it's already a big step in the programming world of Python.

3.3. Exceptions & Traceback

We will apprehend now the exceptions in Python. A way to traceback the errors we can make. In this part, we'll see how Python can intercept errors. It's necessary in some cases to avoid your program to continue with a false result. There are so many options that you can learn here. But I'm not going too far. I will simplify to give the essentials.

Exceptions are the way we intercept the errors. Why do we care? Because it can be interesting to know what kind of errors and where it happened. But also if a user for example type a number instead of a string, we can intercept the error and ask him again.

3.3.1 The basics

The basic error you all know is the division by zero. Impossible in pure computation.

Division by zero

```
1 42/0
```

```
ZeroDivisionError    Traceback (most recent call last)
<ipython-input> in <module>
----> 1 42/0

ZeroDivisionError: division by zero
```

What do we see here ?

1. It writes first the kind of error you did : `ZeroDivisionError`

2. Then you have the Traceback. It will trace you to the error, say you the line, the file, etc. Here we're in a interpreter so it's written `<ipython> in <module>`. But if you where in the file `bar.py`, it would be `bar.py in <module>`

3. Finally there is a few information more about the error. Here, no so much because it's a classic but we can another example.

Error of Type

```
'Hello' + 5
```

```
TypeError   Traceback (most recent call last)
<ipython-input> in <module>
----> 1 'Hello' + 5

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

`TypeError` is a really basic error but here you can see some information about it: `TypeError: can only concatenate str (not "int") to str`. What if you create a script and the error spread all over your functions?

Error through functions

```
def say_hello(name):
    sentence = 'hello ' + name
    print(sentence)

def ask_name_and_say_hello():
    name = input('What\'s your name? ')
    name = int(name) # so the problem won't be the user
       but the script itself. You don't convert name into
        integer.
    say_hello(name)

ask_name_and_say_hello()
```

```
ValueError   Traceback (most recent call last)
<ipython-input> in <module>
      8     say_hello(name)
      9
---> 10 ask_name_and_say_hello()

<ipython-input> in ask_name_and_say_hello()
      6     name = input('What\'s your name? ')
----> 7     name = int(name)
      8     say_hello(name)

ValueError: invalid literal for int() with base 10: '
   Jeremy'
```

You can see here the traceback if from the most recent to the origin of the error. The script is nonsense but you can see Python give you all the traceback. I think the example is relevant and is enough to understand what's happening with all the functions.

3.3.2 Intercept your own errors `try ... except`

You can test some instructions and catch the error if it fails. The syntax is the following one:

- `try:` with instructions indented below
- `expect ...:` with the type of Error an then indented instructions. If you don't specify any Error, it will catch all possible errors. It isn't recommended at all.

When caught, that doesn't mean the script is finished. For example, you can replace the value error with a default value like in the example below. You're just aware that this `ValueError` can happen and you were prepared to bypass the user mistake. You're really a good programmer, congrats!

Intercept an error

```
1 num = input('write an integer :')
2 try:
3     num = int(num)
4
5 except ValueError:
6     print('You enter a name instead of a number, take the
          value 0')
7     num = 0
8
9 print(num)
```

```
write an integer :lala
You enter a name instead of a number, take the value 0
0
```

There's a lot of different errors like `NameError` if your variable you call doesn't exist and `TypeError` when using the wrong type of variable. You will discover by yourself when programming.

3.3.3 `raise` your error

If you want to catch error by yourself, you can also use the keyword raise. You're probably asking why raise our own errors if Python can do it alone? Sometimes, you need to skip some steps or inform your user that there was an error in what they write. It can also be useful to debug more easily if someone is reusing your code for another purposes (like writing a library based on another one). For example, if you want to raise an error if the variable is negative. It's not an error in itself for Python but for your script, it is.

Intercept an error

```
1 i = int(input('Write a number: '))
2 try:
3     if i < 0:
4         raise ValueError
5 except ValueError:
6     print('You have a negative value')
```

```
Write a number: -1
You have a negative value
```

You're now aware of all of these concepts. You're not an expert for sure, we didn't do any practice! That's why I insist for you to do the next exercise seriously.

3.4. Exercise: Karaoke Bar

It's the first time you will create a whole project but we will use several files to build it so you need to open PyCharm to begin your big project! The concept is easy. You're in a karaoke bar (perhaps with a bottle of pastis or anything you judge useful like friends!) and you want to sing some songs and drink some alcohol (not too much).

You have to program the karaoke machine, and also follow the total price for every people in the room. It will be a direct split bill at the end of the evening.

For the karaoke machine, each turn (each time a song is finished), the machine will ask you if you want to play another song or leave the room. If you want another song, it will displays the list available and then you can choose. Caution! You can ask for more than one song at each turn (`*parameters` for example but not compulsory). You can also decide to play a random song from the list (`random.choice()`). ALSO, you can only play maximum 20 songs and after you have to leave.

To order a drink, to test another part of the knowledge you acquire, the barman will come every 3 songs.

Last constraint, you have to use an Exception (`try: / except *****:`) at least once.

Advices

I advise you to divide your code in three different files:

- The file `data.py` with the dictionaries (oh, I helped you) of songs, the price of alcohol, the number max of songs and other variables you find useful...
- The file `functions.py` with all the functions useful for your main algorithm (I advise you to write it on a paper with a pen, all the functions that can be useful...)
- The main file `karaokebar.py` with the main algorithm, the input/output, call for the functions, the while-loop...

Here we are. Now, some things you have to know:

- You have to ask how many people and who enters the room, to save the bill for each person. Due to Covid situation, only 5 people max are allowed.
- You have to remember the waiting list of songs in the machine. It's not always just one song and ask, one song and ask.

- You will need some functions in library like random for `random.choice()` to choose randomly a song in a dictionary. The dictionary has to begin with a number for this function. It also works with a list!
- To spare you time, I already write you a dictionary of 20 songs! What a merciful man!
- I also write a list of alcohol you can order (in two different dictionaries separated between bottle and unit). Be careful, you can decide if just one person pays for the bottle or if you split between everybody. If you do the last thing, you have to round the division at the upper value (`math.ceil(x)` in `math` library or `round()`) to avoid to deal with floats.

data.py

```
# dictionary of songs

songs = {
    1: 'Guys - Chi minii',
    2: 'Aqua - Barbie Girl',
    3: 'Rihanna - Umbrella',
    4: 'Whitney Houston - I will always love you',
    5: 'Michael Jackson - Thriller',
    6: 'Ginjin and Mrs M - Boroo',
    7: 'Clean Bandit - Symphony',
    8: 'Stromae - Papaoutai',
    9: 'Avicii - Levels',
    10: 'Earth Wind and Fire - September',
    11: 'ABBA - Dancing Queen',
    12: 'Haddaway - What is Love',
    13: 'Men at Work - Down Under',
    14: 'Patrick Sebastien - Les Sardines',
    15: 'Beyonce - Love on Top',
    16: 'Clean Bandit - Solo',
    17: 'Sexy Sushi - Cheval',
    18: 'Big Boi - All night',
    19: 'Little Big - Everyday Im drinking',
    20: 'Rick Astley - Never gonna give you up'
}

# dictionary of alcohol

alcohol_unit = {
    'Niislel': 4500,
    'Craft': 5500,
    'Airag': 3500,
    'Eden': 2000
}

alcohol_bottle = {
    'Eden': 40000,
    'Red wine': 60000,
    'White wine': 55000
}
```

4. Object-Oriented Programming

In this chapter, we will study what Python was planned to be : an object-oriented language!

Jérémy, you lost me....
No worries! We will go slowly. Theses lessons about **classes**, **methods** and everything are brand new to you if you didn't study other oriented-object languages.

In this chapter, we will see:

- Open and write into a file: not really into object-oriented stuff but I wanted to see that with you.
- Apprehend **classes** and define **properties** : The really beginning of you being a master of Python ... After this you will conquer the world for sure
- **Specific methods**: A class is not just methods and attributes, you can do a bit more (a lot more actually)
- **Inheritance**: For small projects, a little bit useless but when you will write the whole Windows OS in Python in the last chapter ... (not true, don't panic!). It's a important notion in object-oriented programming that you can find anywhere.

4.1. Open and Write into Files

You'll have to open and write into files a lot of time, to save some data of your script, open data from another software ... When you work with Python, the directory where the script is will be the current directory. To know where you are, you have to use the module `os` and the function `getcwd()`.

Absolute path of your script

```
1 import os
2 print(os.getcwd())
```

```
/home/jeremow/Python_level2
```

Now, if you want to change the folder you're in, there is the function `chdir('/path/to/directory')`. I let you experiment on your own!

Good to know that but how do we open and write into files?
First you have to open the file with Python with the function ... `open()` (not in the `os` module!). It takes two parameters: the path to the file (the file you want to open or create) and how do you want to open it:

- `'r'` : open in read mode. You cannot edit it.
- `'w'` : open in write mode. Be careful, the content of the file will be erased. If the file doesn't exist, Python creates it.
- `'a'` : open in append-writing mode. You'll write at the end of the file without erasing it. If the file doesn't exist, Python creates it.

Opening a file in write mode

```
1 my_file = open('code/3_1/file.txt','w')
```

When you write a path like this `'code/3_1/file.txt'`, it will begin at the current directory (`os.getcwd()`). But if you write like this with a slash as first character, it will be the real path so the real path of your computer. For example, for the same folder as before: `'/home/jeremow/Python_level2/code/3_1'`. It can be dangerous if you want to give your script to someone using real path.

Then you can write in it with the method `write()` and read it with the method ... `read()`. So smart it is. But, only strings! If you want to put numbers, you have to convert it with `str()` to avoid errors. The method `write()` returns the number of characters you wrote.

Write into a file

```
1 my_file.write('Sain uu, I am a real mongolian man ; Hello
      ; Bonjour')
```

```
52
```

To close the file, you use the method `close()`. If another part of your script or another software want to open it but it's opened in your variable `my_file`, you'll get errors.

Close a file

```
1 my_file.close()
```

You cannot read and write at the same time, so you have to close it first and then open it in read mode.

Read the file

```
1 my_file = open('code/3_1/file.txt','r')
2 content = my_file.read()
3 print(content)
4 my_file.close()
```

```
Sain uu, I am a real mongolian man ; Hello ; Bonjour
```

In fact, we don't use so much the method `close()` because we will use another keyword called `with`. I swear to you that's one of the last we will learn! With the `with` statement, you can do operations on file without caring about closing it after. Here is the syntax:

Syntax of `with` statement

```
with open(path_file, opening_mode) as variable_name:
    # operation on the file
```

We have here:

- the keyword `with` to manipulate the file
- the function `open()` which will return the object bound to your file
- `as` to give another name like with modules
- the variable which contains the file as an object.

With a little example:

Example of the syntax `with`

```
with open('code/3_1/file.txt','r') as my_file:
    content = my_file.read()
    print(content)
    my_list = []
    my_list = content.split(';')
    print(my_list)
```

```
Sain uu, I am a real mongolian man ; Hello ; Bonjour
['Sain uu, I am a real mongolian man ', ' Hello ', '
   Bonjour']
```

Python will automatically close the file you wrote in. So even if an error occurs, the file will be closed and you don't have to fear to lose your data. Here's an example with `obspy` library.

WOW JEREMY YOU WILL USE OBSPY FOR THE FIRST TIME WITH US, I'M SO HAPPY.
Yeah I know.

obspy example

```
1  from obspy import read
2
3  st = read('code/3_1/RD.SONA0..SHZ.D.2020.001', 'mseed') #
       class Stream variable
4
5  tr = st[0] # class Trace variable
6  stats_text = str(tr.stats)
7
8  with open('code/3_1/file.txt', 'w') as my_stats:
9      my_stats.write(stats_text)
10
11 with open('code/3_1/file.txt', 'r') as my_stats:
12     print(my_stats.read())
```

```
          network: RD
          station: SONA0
         location:
          channel: SHZ
        starttime: 2020-01-01T00:00:00.000000Z
          endtime: 2020-01-01T23:59:59.980000Z
    sampling_rate: 50.0
            delta: 0.02
             npts: 4320000
            calib: 1.0
          _format: MSEED
            mseed: AttribDict({'dataquality': 'D', '
               number_of_records': 1219, 'encoding': '
               STEIM2', 'byteorder': '>', 'record_length':
               4096, 'filesize': 4993024})
```

Little explanation:

- l.1: First we import the function `read` from obspy library
- l.3: When using `read()`, you can pass the path of your file (here a miniseed). If you precise the type on the second argument, it will be faster for the function to read it. It returns a `Stream` obspy class object which contain all the information about seismic data.
- l.4: Then we take out the first trace. In a stream, you can have different seismic

trace like one on the 1st of January and one on the 13rd but also one from the station SONA0 and one from the station SONA4 for example.

- l.5: Then we get the stats of the trace (sampling rate, station, etc ...) and convert it to text.
- l.7-11: To finish, we write into a text file and recover it afterwards.

If you're interested in saving objects like dictionaries into file, I let you see around the module `pickle` here: https://wiki.python.org/moin/UsingPickle. Be careful with unknown pickle file through internet, never trust it. Only use Pickle file you created yourself.

4.2. Classes and Properties

And here we are. The **Object-Oriented programming**. I see your smile on your face when new notions appear in the lessons. A whole world to discover for some people. It is not just concepts, it's a true philosophy. And in Python, without even knowing about it, **everything is object**. A function, a variable. Behind all of this, the concept of object is here.

You already saw that to add an element in a list, you just have to write: `my_list.append(element)`. And it just seemed a little easier than `append_to_list`. But it's not just aesthetic for the OOP, that's a way of life.

A class is the model to create the object. Inside of it, there will be our **methods**, **attributes**. Attributes are just variables created inside of the object. Do you follow me?

There are a lot of already existing classes like numbers, strings, lists ... but when you create a big script, you'll have to create your own class to make it all a little a bit easy for you, the programmer, but also for the user of your script. If he doesn't understand the logic when he wants to modify something, he will try to call you every two minutes.

4.2.1 Convention PEP8

To write classes, there are some conventions that you can find here: https://www.python.org/dev/peps/pep-0008.

We don't use the *underscore* anymore like for functions. We will use the **Camel Case**. Each time you write a word, you'll use a capital letter. For example, to declare a class with the keyword ... `class`, you'll write: `class NameOfMyClass:`.

4.2.2 Attributes of a class

We will first create a class with some attributes directly taken from seismic data and after we will go further. The main attributes can be find in MiniSeed file in the `stats` of the `Trace`.

`stats` attribute

```
from obspy import read

st = read('code/3_1/RD.SONA0..SHZ.D.2020.001')
tr = st[0]
print(tr.stats)
```

```
         network: RD
         station: SONA0
        location:
         channel: SHZ
       starttime: 2020-01-01T00:00:00.000000Z
         endtime: 2020-01-01T23:59:59.980000Z
   sampling_rate: 50.0
           delta: 0.02
            npts: 4320000
           calib: 1.0
         _format: MSEED
           mseed: AttribDict({'dataquality': 'D', '
              number_of_records': 1219, 'encoding': '
              STEIM2', 'byteorder': '>', 'record_length':
              4096, 'filesize': 4993024})
```

So we will create our class based on the information of *SONA0* the LP and CP stations of Mongolia. As attributes, we take the network, the name of the station, the channel and the sampling rate.

To create an object of the class, we will need a special method called the **instantiation**. So we don't say anymore create an object of the class but create a new instance. The instantiation will create the attributes.

Creation of the class `Station`

```
1  class Station: # define our class Station
2      """
3      Class describing a sensor with attributes:
4      - network
5      - station
6      - channel
7      - sampling_rate
8      """
9
10     def __init__(self): # Our instantiation method
11         """
12         Define the attribute station
13         """
14         self.station = tr.stats.station # from the Trace
               we read before
```

As you can see, the function `__init__(self)` seems classic. The name is `init` and will be the same for every class you write. In the next chapter, we will the see all the special methods, there are always surrounded by two underscores from each side (`__namemethod__`).

In the instantiation, you see the attribute `station`. We create the variable `self.station` inside of the class and we affect the value `SONA0`.

Creation of an instance of `Station`

```
1  my_station = Station()
2  print(my_station)
3  print(my_station.station)
```

```
<__main__.Station object at 0x7f31f4fbeb38>
SONA0
```

The `self.***` is simply the object we create. It's not a new variable, object or anything. So when you write `self.station`, you declare a value inside of a class statement. It will be the same for every attribute and method.

If we put some more things inside of our **instantiator** (= instantiation method), we will have something like below.

More attributes in the class

```
class Station: # define our class Station
    """
    Class describing a sensor with attributes:
    - network
    - station
    - channel
    - sampling_rate
    """

    def __init__(self): # Our instantiation method
        """
        Instantiator of Station class.
        """
        self.network = tr.stats.network # from the Trace
           we read before
        self.station = tr.stats.station
        self.channel = tr.stats.channel
        self.sampling_rate = tr.stats.sampling_rate
```

Now, you can create an object with more attributes.

Instance with several attributes

```
my_station = Station()
print(my_station.network)
print(my_station.channel)
```

```
RD
SHZ
```

If you want to change an attribute, you just have to affect with a new value.

Change an attribute value

```
my_station.sampling_rate = 20.0
print(my_station.sampling_rate)
```

```
20.0
```

What is the point of doing this, all my stations don't have the same name, etc. Yes,

we can have a smart instantiator to declare the attributes when we create the object. We will add parameters to our `__init__` method without forgetting the `self`!

Parameters in the instantiator

```
1  class Station: # define our class Station
2      """
3      Class describing a sensor with attributes:
4      - network
5      - station
6      - channel
7      - sampling_rate
8      """
9
10     def __init__(self, network, station, channel,
           sampling_rate): # Our instantiation method
11         """
12         Instantiator of Station class.
13         """
14         self.network = network # from the Trace we read
               before
15         self.station = station
16         self.channel = channel
17         self.sampling_rate = sampling_rate
```

So when you create a new instance now:

Affecting attributes values when creating an instance

```
1  SONA0_SHZ = Station(tr.stats.network, tr.stats.station,
2                      tr.stats.channel, tr.stats.
                           sampling_rate)
3
4  print(SONA0_SHZ.sampling_rate)
```

```
100.0
```

I highly recommend you to to try to add some others attributes, change the instantiator and everything to really understand what's going on with the classes.

We can also create what's called a **class attribute**. A class attribute is a common attribute to all the instances of your classes. It can be useful to count how many stations you have for example. We define it before the instantiator.

Class attribute

```
class Station: # define our class Station
    """
    Class describing a sensor with attributes:
    - network
    - station
    - channel
    - sampling_rate
    """

    number_of_stations = 0

    def __init__(self, network, station, channel,
       sampling_rate): # Our instantiation method
        """
        Instantiator of Station class.
        """
        Station.number_of_stations += 1

        self.network = network # from the Trace we read
           before
        self.station = station
        self.channel = channel
        self.sampling_rate = sampling_rate
```

Creating several instances

```
SONA0_SHZ = Station(tr.stats.network, tr.stats.station,
                    'SHZ', tr.stats.sampling_rate)
SONA0_SHE = Station(tr.stats.network, tr.stats.station,
                    'SHE', tr.stats.sampling_rate)
SONA0_SHN = Station(tr.stats.network, tr.stats.station,
                    'SHN', tr.stats.sampling_rate)

print(SONA0_SHN.number_of_stations)
```

```
3
```


4.2.3 Methods of a class

The **attributes** are the variables attached to the class and the **methods** are the actions. It is just functions who manipulate the objects of the class like `append` for the class `list`.

We take again our `Station` class and we will add the dedicated function to plot the trace of it inside of `obspy`. I know, it's not really useful, but it's to learn to you `obspy` and the creation of classes at the same time.

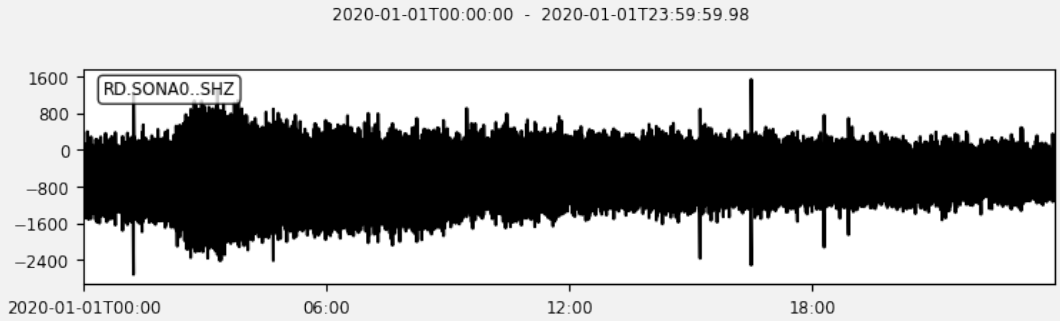
We will rewrite our class `Station` to build new instance with just the trace of one trace. And then we will add the feature to plot it.

Method of a class

```
1  class Station: # define our class Station
2      """
3      Class describing a sensor with attributes:
4      - Trace related to obspy file
5      - network
6      - station
7      - channel
8      - sampling_rate
9      """
10
11     number_of_stations = 0
12
13     def __init__(self, trace): # Our instantiation method
14         """
15         Instantiator of Station class.
16         """
17         Station.number_of_stations += 1
18
19         self.trace =  trace
20         self.network = trace.stats.network # just an
               artefact to access easily
21         self.station = trace.stats.station
22         self.channel = trace.stats.channel
23         self.sampling_rate = trace.stats.sampling_rate
24
25     def plot(self):
26         """
27         Plot the trace inside of our class.
28         """
29         self.trace.plot()
```

Test of the `plot` method

```
1 SONA0_SHZ = Station(tr) # tr was the trace #0 of SONA0
    SHZ
2 SONA0_SHZ.plot()
```



Et voilà! You wrote your first method. Now you can imagine all what you want with it. For sure, it doesn't seem really useful to create another thing similar to what `obspy` can offer. But if you work on a big research project and want to do use your functions with a formatted number of parameters of an obspy `Stream` object, create your own class will help you to speed up your efficiency.

4.2.4 A bit more about `self`

Just a little thing with the `self`. I said to you that it call the object. When you call attributes, it will search for it in the object you created but when you call functions, it will search for it in your written class. So with our example, calling `SONA0_SHZ.plot()` is exactly the same as `Station.plot(SONA0_SHZ)`. And here is the `self`.

4.2.5 To go further

There are a lot of others things to know about classes like `classmethod`, `staticmethod`, the function `dir`, the dictionary `__dict__` but I don't want you to have mix feelings about it. Especially because you won't use it. You can learn by yourself if you're interested.

4.2.6 Properties

The classes are the first point of the object-oriented programming. But in Python like in others languages, there are some concepts which created the philosophy of the `object`.

Encapsulation

Encapsulation is the principle to hide or protect some data of your object. It's not compulsory to use this at the level we are but I want to present to you just the basics of it if one day, you need it.

How to change or get the attributes so? With **Accessor** (getter) or **Mutator** (setter). So when you want to have access to your attribute, instead of writing `my_station.network`, you'll write `my_station.get_network()`. You use a method. And the same way to affect a new value: `my_station.set_network(new_value)`.

The drawback is it's really long to write like this. So if you don't need it, don't use it. And here comes the `property`. You can manage an attribute with three functions. `_get_***`, `_set_***` and `_del_***`. I will write an example with our `Station` and you will understand.

Property in a class

```
1  class Station: # define our class Station
2      """
3      Class describing a sensor with attribute station
4      """
5
6      def __init__(self, station):
7          """
8          Instantiator of Station class
9          """
10         self._station = station
11
12     def _get_station(self):
13         """
14         Method to read the variable station
15         """
16         print('I return you the value of the station')
17         return self._station
18
19     def _set_station(self, new_station):
20         """
21         Method to set a new value for the station
22         """
23         print('You modify the station from {} to {}'.
               format(self._station, new_station))
24         self._station = new_station
25
26     station = property(_get_station,_set_station)
```

Now if you create an new instance of the `Station` and after you call or modify the **attribute** `station`, you'll get what you wrote inside of the **method** `_ get_station` or `_set_station`.

Example of an accessor with `Station`

```
1 SONA0 = Station('SONA0')
2 print(SONA0.station)
```

```
I return you the value of the station
SONA0
```

Example of a mutator with `Station`

```
1 SONA0.station = 'SONA1'
2 print(SONA0.station)
```

```
You modify the station from SONA0 to SONA1
I return you the value of the station
SONA1
```

You see some difference like the underscore of `_station`. That will be variable protected from the outside. So, now when you want to access the value, it will call the function you pass into the first parameter of `property`. And to set the value, it will call the second one. This way of programming allows you to avoid the user of your class to change attributes that could mess your script.

If it's not clear, just analyze the code and try to understand.

It doesn't mean that you cannot write and read directly through `SONA0._name` but it's the convention for Python to consider the attribute as protected. And when you are outside of a class, you know that if this attribute of your object is written like this, you don't have to use it directly, especially modify it directly. Else terrible thing could happen inside of complex classes which needed this attribute.

4.3. Special Methods

What it that? We already saw one, it was the `__init__` function of our class which is the instantiator. But in fact, there are much more than you can imagine. I will explain the main ones but not too much. If you're interested in, you'll find by yourself the possibilities.

I think you understood but these methods are always written with two underscores before and after.

4.3.1 Representation of your object

When you give your object to the interpreter or to the `print` function, if you didn't define anything, you'll have the *ugly* version:

Default representation of a class

```
1 class Station: # definition of the class
2     def __init__(self, station):
3         self.station = station
4
5 # Test
6 SONA0 = Station('SONA0')
7 print(SONA0)
8 SONA0
```

```
<__main__.Station object at 0x7f3f71d2ca90>
<__main__.Station object at 0x7f3f71d2ca90>
```

Useful information but not for you, so you can interact with it to give you what's interesting about object with the special method `__repr__` if you just write in the interpreter. To modify the behavior of `print`, you have to create the method `__str__`.

Changing the behavior of the representation of a class

```
1  class Station:
2      def __init__(self, station):
3          self.station = station
4
5      def __repr__(self):
6          return "My station is called: {} (without print)"
               .format(self.station)
7
8      def __str__(self):
9          return "My station is called {} (with print)".
               format(self.station)
10
11 # Test
12 SONA0 = Station('SONA0')
13 print(SONA0)
14 SONA0
```

```
My station is called SONA0 (with print)
My station is called: SONA0 (without print)
```

4.3.2 Access to the attributes of your class

If you want to play with the attributes of your class, you'll have to modify the function `__getattr__`, `__setattr__` or `__delattr__`. It allows you to add some attributes to your classes or print something if the attribute doesn't exist or even delete some attributes... you can also modify it to avoid someone to do this in the class. I don't explain more, do research if you want!

4.3.3 Mathematics methods

To manipulate time for example, you need to do modify how the `+` and other signs will work because it's not the same as with number. We take the example of a *time* class.

A `Time` class

```
1  class Time:
2      """Class for time"""
3
4      def __init__(self, min=0, sec=0):
5          """Instantiator of the class"""
6          self.min = min # Nb of minutes
7          self.sec = sec # Nb of seconds
8      def __str__(self):
9          """Display in a beautiful way"""
10         return "{0:02}:{1:02}".format(self.min, self.sec)
11
12 # Test
13 t1 = Time(3, 5)
14 print(t1)
```

```
03:05
```

Now if you want to add four seconds to your time `t1`.

Type Error with `Time`

```
1  t1 + 4
```

```
TypeError     Traceback (most recent call last)
<ipython-input> in <module>
----> 1 t1 + 4

TypeError: unsupported operand type(s) for +: 'Time' and '
   int'
```

Oh, it's an error. Yes, Python doesn't know how to add your `Time` class and an integer... like the `str` class if you remember.So you have to write it with `__add__` special method.

Mathematical special method

```
1  class Time:
2      """Class for time"""
3
4      def __init__(self, min=0, sec=0):
5          """Instantiator of the class"""
6          self.min = min # Nb of minutes
7          self.sec = sec # Nb of seconds
8
9      def __str__(self):
10         """Display in a beautiful way"""
11         return "{0:02}:{1:02}".format(self.min, self.sec)
12
13     def __add__(self, object_to_add):
14         """The object to add is an integer, number in
               seconds"""
15         new_time = Time()
16         # We copy self in the created object to have the
               same time
17         new_time.min = self.min
18         new_time.sec = self.sec
19         # We add the new time
20         new_time.sec += object_to_add
21         # if the new time in seconds >= 60
22         if new_time.sec >= 60:
23             new_time.min += new_time.sec // 60
24             new_time.sec = new_time.sec % 60
25         # we return the new Time
26         return new_time
```

Test of `Time` class

```
1  t1 = Time(3, 5)
2  print(t1)
3
4  t1 = t1 + 56
5  print(t1)
```

```
03:05
04:01
```


That's for `add` but there are also for everything else:

- `__sub__`: for `-`
- `__mul__`: for `*`
- `__truediv__`: for `/`
- `__floordiv__`: for `//`
- `__mod__`: for `%` (modulo)
- `__pow__`: for `**` (power)

And what about `t1 = 56 + t1`. You have the eyes Sherlock. It doesn't work ... To avoid this problem, you have to declare `__radd__` like this:

Reverse add

```
1     def __radd__(self, object_to_add):
2         return self + object_to_add
```

If you want to use `+=` you have to redefine `__iadd__`, etc. Other useful special methods to change are the comparison operators: https://blog.cambridgespark.com/magic-methods-a8d93dc55012

4.4. Inheritance

What is inheritance? **Inheritance** is one of the most used concepts in object-oriented programming. At first you'll ask yourself *What the hell am I learning?* and after you'll be like *Why the hell am I learning this?* and finally *Why the hell didn't know about that before*?

And here the guide to go through it ... we will begin with simple examples.

4.4.1 The concept

Inheritance is a concept where you can declare that one of your class will be built on the model of another one which is called the **Base Class**. If a class `Child` inherits from the class `Mother`, the objects created on the model of the `Child` will have access to the **methods** and **attributes** of the class `Mother`. This `Child`class is called **Derived Class**.

The goal is to add some features to our Base Class like other methods and attributes which will get some great make up on it. The Derived Class can also redefined some methods in the Base Class to adapt to your personal use (print other stuffs, change figures ...).

But let's start at the beginning with an example. We have a class `Animal` where we can create ... animals. When we define an animal, they have attributes (like the diet: meat or plants) and methods (like eat, drink, shout).

Now we can define the class `Dog` which inherits from the `Animal` class, so it has its methods and attributes. Why `Dog` from `Animal` and not the contrary:

- `Dog` inherits from `Animal` because a dog is an animal.
- `Animal` doesn't inherit from `Dog` because an animal is not a dog.

With the same model : a car is a vehicle but all vehicles are not cars. So a `Car` can inherits from `Vehicle`. Now we can write a little bit of code!

4.4.2 The single inheritance

We oppose **single inheritance** to **multiple inheritance** that we won't see together.

Here's the syntax. We will define a `BaseClass` and a `DerivedClass` which inherits from `BaseClass`.

Inheritance syntax

```
class BaseClass:
    """
    Base Class to draw a picture of inheritance
    """
    def my_method(self):
        pass
    pass # word to let something empty

class DerivedClass(BaseClass):
    """
    Derived Class which inherits from BaseClass.
    Same methods and attributes.
    Even if there is none here.
    """

    pass
```

So you have the syntax to create inheritance between two classes. So, you write methods and attributes into `BaseClass` and then you create an instance the `DerivedClass`. If you call a method of the object:

Calling a method not defined in `DerivedClass`

```
my_object = DerivedClass()
my_object.my_method()
```

If the method doesn't exist in `DerivedClass`, Python will search for it in the `BaseClass`. But it's not that easy to understand and with a concrete example, you'll understand what I mean.

Calling a method not defined in `DerivedClass`

```
class Person:
    """Class for a person"""
    def __init__(self, name):
        """Instantiator"""
        self.name = name
        self.surname = "Jeremy"
    def __str__(self):
        """Method to print our object in a beautiful way
           """
        return "{} {}".format(self.surname, self.name)

class Seismologist(Person):
    """Class to define a seismologist.
    Inherits from Person"""

    def __init__(self, name, position):
        """A Seismologist is defined with his name and
           position"""
        self.name = name
        self.position = position
    def __str__(self):
        """Method to print our object in a beautiful way
           """
        return "Name: {0}, Position: {1}".format(self.
           name, self.position)
```

So now, we have a single inheritance. But if we create a `Seismologist`, you'll have some problems ... let's see together.

Error with instantatior

```
1 jeremy = Seismologist("Hraman", "cooperant")
2
3 print(jeremy.name)
4 print(jeremy)
5 print(jeremy.surname)
```

```
Hraman
Name: Hraman, position cooperant

--------------
AttributeError    Traceback (most recent call last)
<ipython-input> in <module>
      3 print(jeremy.name)
      4 print(jeremy)
----> 5 print(jeremy.surname)

AttributeError: 'Seismologist' object has no attribute '
   surname'
```

Why the surname doesn't exist even if we defined it in the class `Person`?
Because when we wrote our two classes, we defined the method `__init__` in both of it. So when you create an object of `Seismologist`, it will call the instantiator `__init__` from the class `Seismologist` and not the `Person` one. HOWEVER, the attribute `surname` is in defined only in the instantiator of the class `Person`, so you don't have access to it.
So, the way to avoid this problem is to call the instantiator of the class `Person` when calling the instantiator of the class `Seismologist`. To do so, you could call the instantatior of `Person` like this: `Person.__init(name)__`. But we will use the function `super()` which will automatically call the **Base Class** of `Seismologist` without even caring about the name of the Base Class: `super().__init(name)__`. I think it's a little blurry in your head but watch to this example.

Function `super()`

```
class Person:
    """Class for a person"""
    def __init__(self, name):
        """Instantiator"""
        self.name = name
        self.surname = "Jeremy"
    def __str__(self):
        """Method to print our object in a beautiful way
           """
        return "{} {}".format(self.surname, self.name)

class Seismologist(Person):
    """Class to define a seismologist.
    Inherits from Person"""

    def __init__(self, name, position):
        """A Seismologist is defined with his name and
           position"""
        super().__init__(name)
        self.position = position
    def __str__(self):
        """Method to print our object in a beautiful way
           """
        return "Name: {0}, position {1}".format(self.name
           , self.position)
```

So you see we even removed the `self.name = name` from the instantiator of the `Seismologist` class because it's defined in the `Person` instantiator. Let's run this code.

Test of the `super()` in instantiator

```
jeremy = Seismologist("Hraman", "cooperant")

print(jeremy.name)
print(jeremy)
print(jeremy.surname)
```

```
Hraman
Name: Hraman, position cooperant
Jeremy
```

It's magic, right? If it's not so clear in your head, try to read it again, change methods, call new attributes ...

4.4.3 Verify an instance and a subclass

Two functions exist and are useful to verify if an object belongs to a class (`isinstance`) and if a class inherits from another one (`issubclass`).

issubclass

```
1 print(issubclass(Seismologist, Person))
2 print(issubclass(Seismologist, object))
3 print(issubclass(Person, object))
4 print(issubclass(Person, Seismologist))
```

```
True
True
True
False
```

Yeah, I didn't say to you, but as a object-oriented language, ALL is `object` in Python. So everything derives from it and inherits from it.

isinstance

```
1 print(isinstance(jeremy, Seismologist))
2 print(isinstance(jeremy, Person))
3 print(isinstance(jeremy, object))
4 # please, I'm not just an object, I have a soul
```

```
True
True
True
```

5. ObsPy

Obspy is the worldwide library for seismic data manipulation in Python. It includes nearly everything you need to work properly at the institute.

If you want to search for something into Obspy, the website is really well documented (http://docs.obspy.org/) and you will find for sure the love of your life on it. It is a bit like the Tinder of seismologist.

This chapter is mostly based and copied from MESS 2014 (https://github.com/obspy/mess2014-notebooks/) and Seismo-Live (https://krischer.github.io/seismo_live_build/tree/index.html)

The core functionality of ObsPy is provided by:

- The most important base classes:
  - the `UTCDateTime` class handles time information
  - the `Stream/Trace` classes handle waveform data
  - the `Catalog/Event/`... classes handle event metadata (modelled after QuakeML)
  - the `Inventory/Station/Response/`... classes handle station metadata (modelled after FDSN StationXML)
- And the associated functions:
  - The `read` function. Reads all kinds of waveform file formats. Outputs a `Stream` object.
  - The `read_events` function. Reads QuakeML (and MCHEDR) files. Outputs a `Catalog` object.
  - The `read_inventory` function. Reads FDSN StationXML files. Outputs an `Inventory` object.
- the most important classes/functions can be imported from main namespace (from obspy import ...)
- Unified interface and functionality for handling waveform data regardless of data source

- `read`, `read_events` and `read_inventory` functions access the appropriate file-format submodule/plugin using filetype autodiscovery
- `obspy.core.util` includes some generally useful utility classes/functions (e.g. for geodetic calculations, Flinn-Engdahl regions, ...)
- some convenience command line scripts are also included (e.g. `obspy-plot`, `obspy-print`, `obspy-scan`, ..).

5.1. Timestamps and File Format

5.1.1 Handling Timestamps

This is a bit dry to learn but not very difficult and important to know. It is used everywhere in ObsPy!

- All absolute time values are consistently handled with this class
- Based on a double precision POSIX timestamp for accuracy
- Timezone can be specified at initialization (if necessary)
- In Coordinated Universal Time (UTC) so no need to deal with timezones, daylight savings, ...

Use of `UTCDateTime`

```
1 from obspy import UTCDateTime
2
3 # mostly time strings defined by ISO standard
4 print(UTCDateTime("2011-03-11T05:46:23.2"))
5  # non-UTC timezone input
6 print(UTCDateTime("2011-03-11T14:46:23.2+09:00"))
7 print(UTCDateTime(2011, 3, 11, 5, 46, 23, 200000))
8 print(UTCDateTime(1299822383.2))
```

```
2011-03-11T05:46:23.200000Z
2011-03-11T05:46:23.200000Z
2011-03-11T05:46:23.200000Z
2011-03-11T05:46:23.200000Z
```

Current time can be initialized by leaving out any arguments.

Current time with `UTCDateTime`

```
1 print(UTCDateTime())
```

```
2021-08-20T02:04:03.090280Z
```

If you want to access to the attributes of the variable `time`:

Attributes of `UTCDateTime` object

```
time = UTCDateTime()
print(time.year)
print(time.julday)
print(time.timestamp)
print(time.weekday)
```

```
2021
232
1629427240.3664742
4
```

There's more and you can know which one by writing in your software `time.` and pressing `Tab` on your keyboard.

To do some operations with `time`, you have to use the **second** as the quantity. You can **add or subtract** seconds from one `UTCDatetime` object BUT you can only do the **difference** between two `UTCDateTime` objects.

The result of the first one will be a **delta** in seconds and the other will be an `UTCDateTime` object.

Attributes of `UTCDateTime` object

```
from time import sleep

t1 = UTCDateTime()
sleep(5) # stop the script for 5 seconds
t2 = UTCDateTime()

print('t1:', t1, 'and t2:', t2)
print('t1 + 1 hour:', t1 + 3600) # one hour
print('t1 - 1 hour:', t1 - 3600)
print('Delta between t2 and t1:', t2 - t1, 'seconds')
```

```
t1: 2021-08-20T02:55:53.796493Z and t2: 2021-08-20T02
   :55:58.801911Z
t1 + 1 hour: 2021-08-20T03:55:53.796493Z
t1 - 1 hour: 2021-08-20T01:55:53.796493Z
Delta between t2 and t1: 5.005418 seconds
```

5.1.2 Handling File Format

SEED Identifiers

According to the SEED standard (http://www.fdsn.org/pdf/SEEDManual_V2.4.pdf), which is fairly well adopted, the following nomenclature is used to identify seismic receivers:

- **Network code**: Identifies the network/owner of the data. Assigned by the FDSN and thus unique
- **Station code**: The station within a network. NOT UNIQUE IN PRACTICE! Always use together with a network code
- **Location ID**: Identifies different data streams within one station. Commonly used to logically separate multiple instruments at a single station
- **Channel codes**: Three character code:
    1. Band and approximate sampling rate
    2. The type of instrument
    3. The orientation

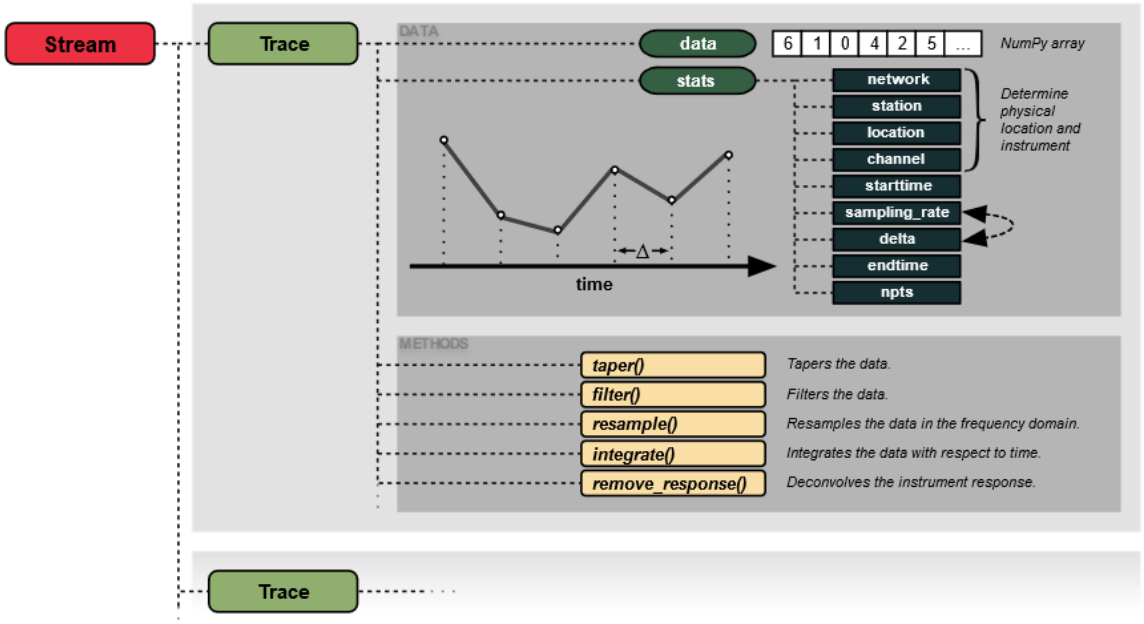
This results in full ids of the form **NET.STA.LOC.CHAN**, e.g. **RD.SONA0..SHZ**.

In seismology we generally distinguish between three separate types of data:

1. **Waveform Data** - The actual waveforms as time series
2. **Station Data** - Information about the stations' operators, geographical locations, and the instrument's responses
3. **Event Data** - Information about earthquakes

Some formats have elements of two or more of these.

Waveform Data



There are a myriad of waveform data formats but in Europe and the USA two formats dominate: **MiniSEED** and **SAC**.

MiniSEED

- This is what you get from datacenters and also what they store, thus the original data
- Can store integers and single/double precision floats
- Integer data (e.g. counts from a digitizer) are heavily compressed: a factor of 3-5 depending on the data
- Can deal with gaps and overlaps
- Multiple components per file
- Contains only the really necessary parameters and some information for the data providers

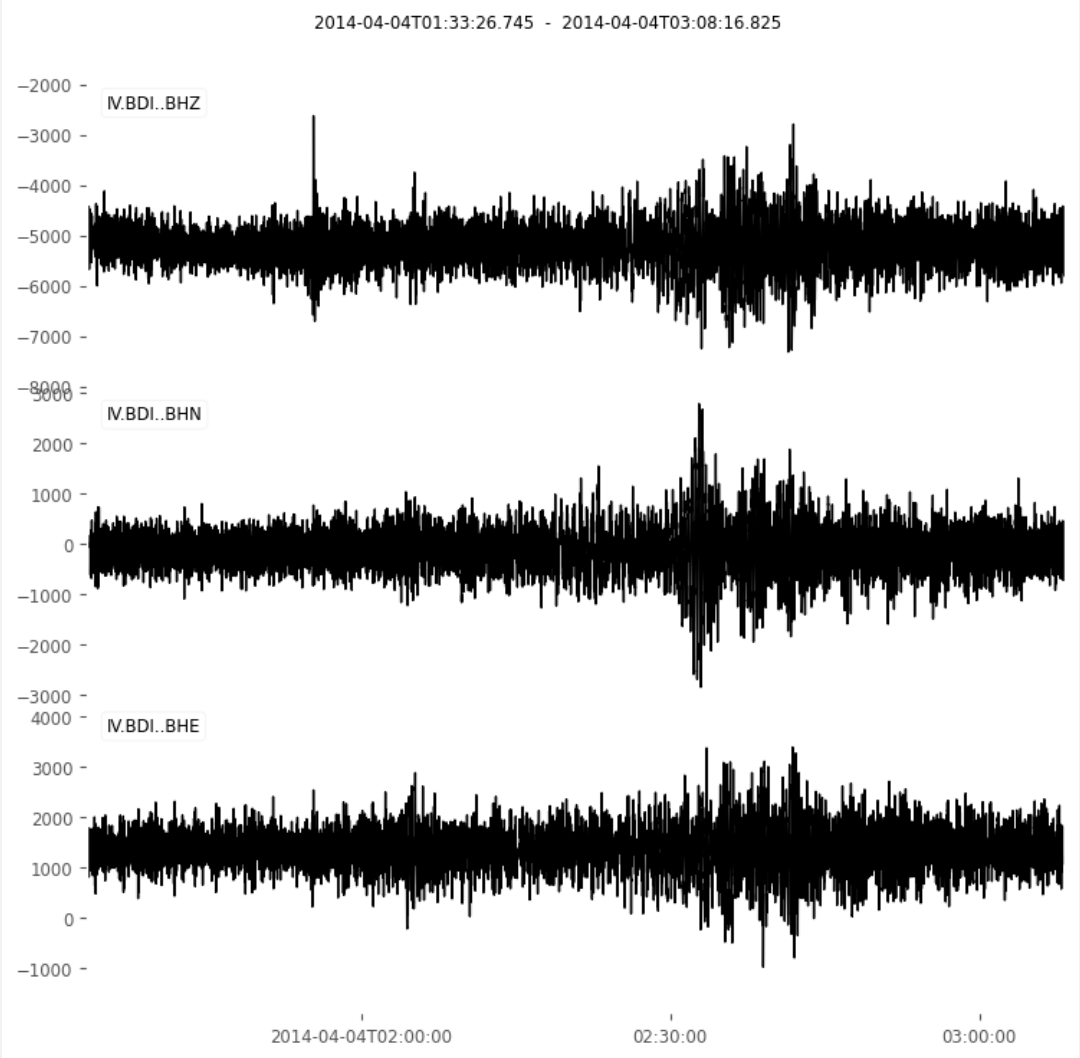
Opening a MiniSEED and print its information

```
1  from obspy import read
2
3  # ObsPy automatically detects the file format.
4  st = read("data/example.mseed") # STREAM
5  print(st)
6
7  tr = st[0] # TRACE
8
9  # Fileformat specific information is stored here.
10 print(tr.stats)
```

```
6 Trace(s) in Stream:
IV.BDI..BHE | 2014-04-04T01:33:37.045000Z - 2014-04-04T02
   :15:10.695000Z | 20.0 Hz, 49874 samples
IV.BDI..BHE | 2014-04-04T02:15:23.535000Z - 2014-04-04T03
   :08:04.485000Z | 20.0 Hz, 63220 samples
IV.BDI..BHN | 2014-04-04T01:33:43.975000Z - 2014-04-04T02
   :15:12.125000Z | 20.0 Hz, 49764 samples
IV.BDI..BHN | 2014-04-04T02:15:22.725000Z - 2014-04-04T03
   :08:10.025000Z | 20.0 Hz, 63347 samples
IV.BDI..BHZ | 2014-04-04T01:33:26.745000Z - 2014-04-04T02
   :15:11.195000Z | 20.0 Hz, 50090 samples
IV.BDI..BHZ | 2014-04-04T02:15:24.025000Z - 2014-04-04T03
   :08:16.825000Z | 20.0 Hz, 63457 samples
         network: IV
         station: BDI
        location:
         channel: BHE
       starttime: 2014-04-04T01:33:37.045000Z
         endtime: 2014-04-04T02:15:10.695000Z
   sampling_rate: 20.0
           delta: 0.05
            npts: 49874
           calib: 1.0
         _format: MSEED
           mseed: AttribDict({'dataquality': 'D', '
              number_of_records': 108, 'encoding': 'STEIM2
              ', 'byteorder': '>', 'record_length': 512, '
              filesize': 374272})
```

Plotting the seismic data

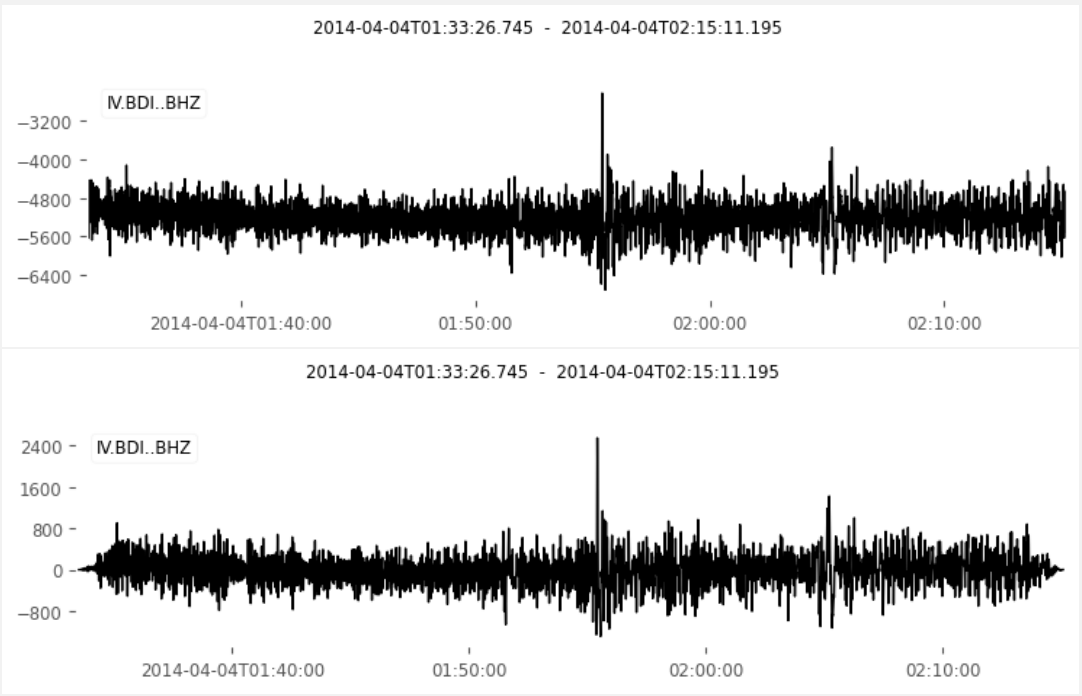
```
1 st.plot()
```



Some basics operation on data

```
# This is a quick interlude to teach you basics about how to work with Stream/Trace objects.
# We need to copy because some actions are modifying the original data
st2 = st.copy()

# To use only part of a Stream, use the select() function.
st2 = st2.select(component="Z")
print(st2)

# Stream objects behave like a list of Trace objects.
tr = st2[0]
print(tr.stats)
tr.plot()

# Some basic processing. Please note that these modify the existing object.
tr.detrend("linear")
tr.taper(type="hann", max_percentage=0.05)
tr.filter("lowpass", freq=0.5)
tr.plot()
```

```
2 Trace(s) in Stream:
IV.BDI..BHZ | 2014-04-04T01:33:26.745000Z - 2014-04-04T02
   :15:11.195000Z | 20.0 Hz, 50090 samples
IV.BDI..BHZ | 2014-04-04T02:15:24.025000Z - 2014-04-04T03
   :08:16.825000Z | 20.0 Hz, 63457 samples
         network: IV
         station: BDI
        location:
         channel: BHZ
       starttime: 2014-04-04T01:33:26.745000Z
         endtime: 2014-04-04T02:15:11.195000Z
   sampling_rate: 20.0
           delta: 0.05
            npts: 50090
           calib: 1.0
         _format: MSEED
           mseed: AttribDict({'dataquality': 'D', '
              number_of_records': 105, 'encoding': 'STEIM2
              ', 'byteorder': '>', 'record_length': 512, '
              filesize': 374272})
```



You can write it again by simply specifying the format.

Saving into a new file

```
st.write("temp.mseed", format="mseed")
```

SAC

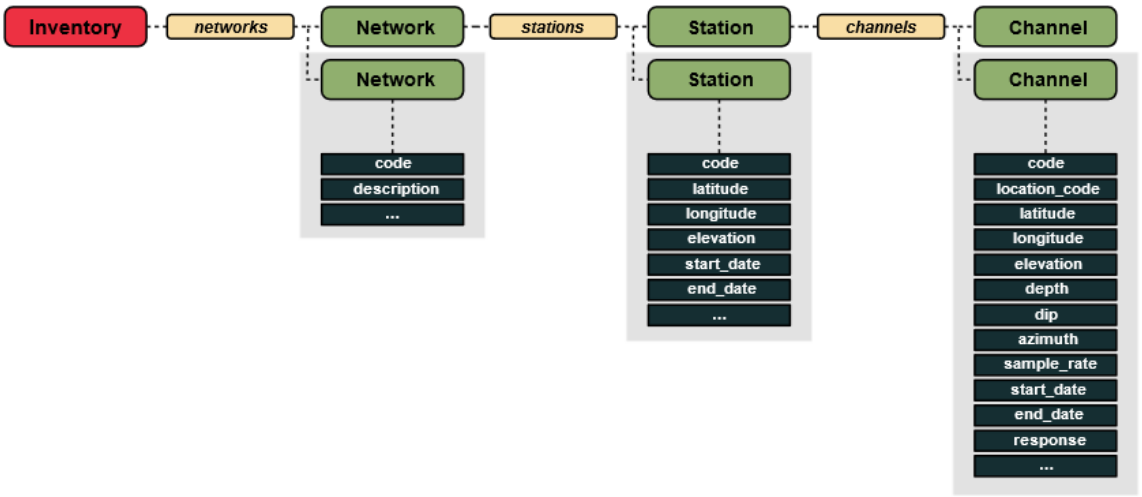
- Custom format of the `sac` code
- Simple header and single precision floating point data
- Only a single component per file and no concept of gaps/overlaps
- Used a lot due to `sac` being very popular and the additional basic information that can be stored in the header

Sac format

```
1 st = obspy.read("data/example.sac") + obspy.read("data/
    waveform_BFO_BHE.sac")
2 print(st)
3 print(st[0].stats)
```

```
2 Trace(s) in Stream:
AI.BELA..BHN | 2010-03-11T06:17:18.000000Z - 2010-03-11T08
   :02:18.000000Z | 20.0 Hz, 126001 samples
GR.BFO..BHE  | 2011-03-11T05:46:23.021088Z - 2011-03-11T06
   :36:22.971088Z | 20.0 Hz, 60000 samples

         network: AI
         station: BELA
        location:
         channel: BHN
       starttime: 2010-03-11T06:17:18.000000Z
         endtime: 2010-03-11T08:02:18.000000Z
   sampling_rate: 20.0
           delta: 0.05
            npts: 126001
           calib: 1.0
         _format: SAC
             sac: AttribDict({'delta': 0.050000001, 'b':
                0.0, 'e': 6300.0, 'internal0': 2.0, 'stla
                ': -77.875, 'stlo': -34.6269, 'stel':
                262.0, 'stdp': 0.0, 'cmpaz': 0.0, 'cmpinc
                ': 90.0, 'nzyear': 2010, 'nzjday': 70, '
                nzhour': 6, 'nzmin': 17, 'nzsec': 18, '
                nzmsec': 0, 'nvhdr': 6, 'norid': 0, 'nevid
                ': 0, 'npts': 126001, 'iftype': 1, 'idep':
                 5, 'leven': 1, 'lpspol': 0, 'lcalda': 1,
                'unused23': 0, 'kstnm': 'BELA', 'khole':
                '', 'kcmpnm': 'BHN', 'knetwk': 'AI', '
                kevnm': ''})
```

Station Data



Station data contains information about the organization that collections the data, geographical information, as well as the instrument response. It mainly comes in three formats:

1. `(dataless) SEED`: Very complete but pretty complex and binary. Still used a lot, e.g. for the Arclink protocol
2. `RESP`: A strict subset of SEED. ASCII based. Contains ONLY the response.
3. `StationXML`: Essentially like SEED but cleaner and based on XML. Most modern format and what the datacenters nowadays serve. Use this if you can.

ObsPy can work with all of them but today we will focus on StationXML. They are `XML` files:

StationXML file

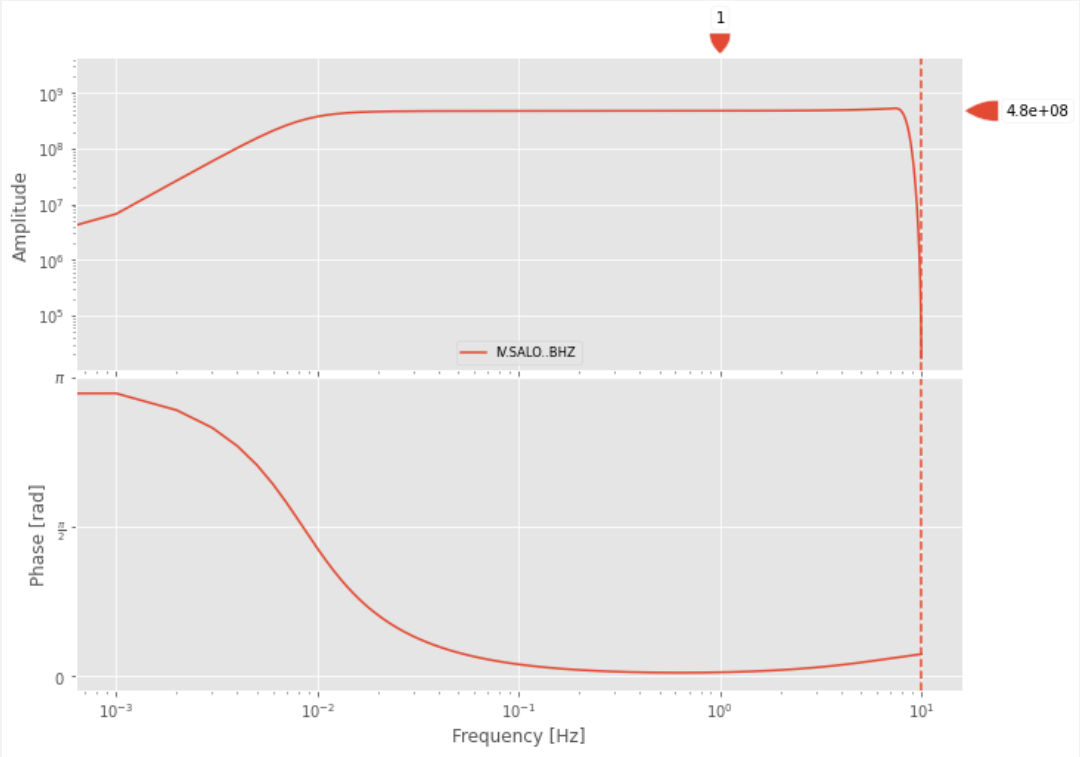
```
1 from obspy import read_inventory
2
3 # Use the read_inventory function to open them.
4 inv = read_inventory("data/all_stations.xml")
5 print(inv)
```

```
Inventory created at 2015-05-13T23:31:49.000000Z
    Sending institution: SeisComP3 (INGV)
    Contains:
        Networks (2):
            IV, MN
        Stations (8):
            IV.BDI (BAGNI DI LUCCA)
            IV.BOB (BOBBIO)
            IV.CAVE (CAVE)
            IV.PESA (PESARO)
            IV.PRMA (Parma)
            IV.SALO (Salo)
            IV.TEOL (Teolo)
            MN.VLC (Villacollemandina, Italy)
        Channels (40):
            IV.BDI..BHZ, IV.BDI..BHN, IV.BDI..BHE, IV.BDI
               ..HHZ, IV.BDI..HHN,
            IV.BDI..HHE, IV.BOB..BHZ, IV.BOB..BHN, IV.BOB
               ..BHE, IV.BOB..HHZ,
            IV.BOB..HHN, IV.BOB..HHE, IV.CAVE..HHZ, IV.
               CAVE..HHN, IV.CAVE..HHE
            IV.PESA..BHZ, IV.PESA..BHN, IV.PESA..BHE, IV.
               PESA..HHZ,
            IV.PESA..HHN, IV.PESA..HHE, IV.PRMA..BHZ, IV.
               PRMA..BHN,
            IV.PRMA..BHE, IV.PRMA..HHZ, IV.PRMA..HHN, IV.
               PRMA..HHE,
            IV.SALO..BHZ, IV.SALO..BHN, IV.SALO..BHE, IV.
               SALO..HHZ,
            IV.SALO..HHN, IV.SALO..HHE, IV.TEOL..BHZ, IV.
               TEOL..BHN,
            IV.TEOL..BHE, IV.TEOL..HHZ, IV.TEOL..HHN, MN.
               VLC..BHZ, MN.VLC..BHN
```


You can see that they can contain an arbitrary number of networks, stations, and channels. You can plot the instrument response:

StationXML instrument response

```
1 inv.select(network="IV", station="SALO", channel="BHZ").
    plot_response(0.001)
```



Coordinates of single channels can also be extracted. This function also takes a `date time` argument to extract information at different points in time.

How to get coordinates of a station

```
1 inv.get_coordinates("IV.SALO..BHZ")
```

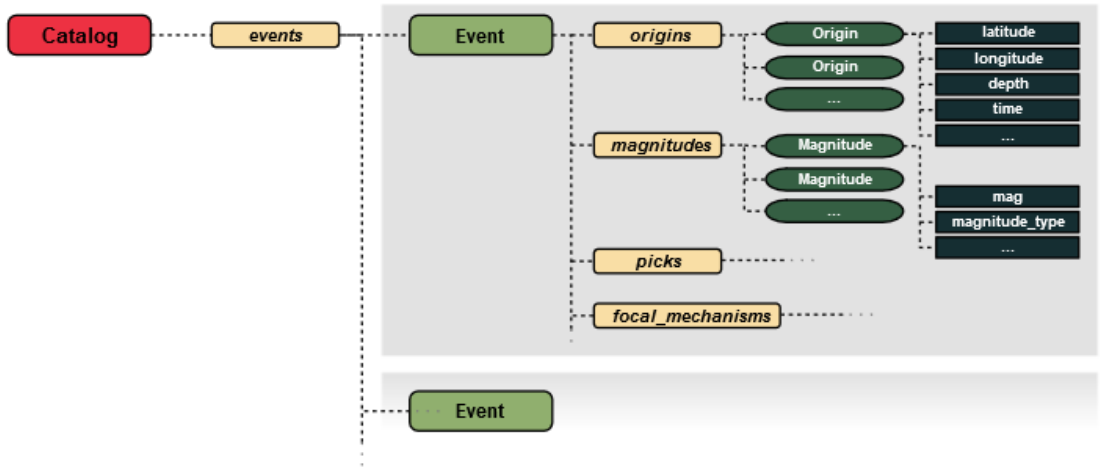
```
{'latitude': 45.6183,
 'longitude': 10.5243,
 'elevation': 600.0,
 'local_depth': 1.0}
```


And it can naturally be written again, also in modified state.

Write a XML file

```
1 inv.select(channel="BHZ").write("data/temp.xml", format="
    stationxml")
```

Event Data



Event data is essentially served in either very simple formats like NDK or the CMTSOLUTION format used by many waveform solvers:

Datacenters on the hand offer QuakeML files, which are surprisingly complex in structure but can store complex relations. To read it, you use the `read_events` function of Obspy.

Read events of a QuakeML file

```
1 cat = obspy.read_events("data/GCMT_2014_04_01__Mw_8_1.xml
    ") + obspy.read_events("data/event_tohoku_mainshock.
    xml")
2 print(cat)
3 print(cat[0])
```

```
2 Event(s) in Catalog:
2014-04-01T23:47:31.500000Z | -19.700,  -70.810 | 8.1 Mwc
2011-03-11T05:46:23.200000Z | +38.296, +142.498 | 9.1 MW
```

```
Event:  2014-04-01T23:47:31.500000Z | -19.700,  -70.810 |
   8.1 Mwc

                     resource_id: ResourceIdentifier(id="
                        smi:service.iris.edu/fdsnws/event
                        /1/query?eventid=4597319")
                      event_type: 'earthquake'
             preferred_origin_id: ResourceIdentifier(id="
                smi:www.iris.edu/spudservice/momenttensor
                /gcmtid/C201404012346A#cmtorigin")
     preferred_focal_mechanism_id: ResourceIdentifier(id="
        smi:ds.iris.edu/spudservice/momenttensor/gcmtid/
        C201404012346A/quakeml#focalmechanism")
                           ---------
            event_descriptions: 1 Elements
              focal_mechanisms: 1 Elements
                       origins: 2 Elements
                    magnitudes: 1 Elements
```

5.1.3 Handling Waveform Data

We already saw it but to open a file you have to write:

Open a stream

```
1 from obspy import read
2 st = read("./data/waveform_PFO.mseed", format="mseed")
3 print(st)
```

```
2 Trace(s) in Stream:
II.PFO.00.BHZ | 2011-03-11T05:46:23.019500Z - 2011-03-11
   T06:36:22.969500Z | 20.0 Hz, 60000 samples
II.PFO.10.BHZ | 2011-03-11T05:46:23.019500Z - 2011-03-11
   T06:36:22.994500Z | 40.0 Hz, 120000 samples
```

- UNIX wildcards can be used to read multiple files simultaneously
- Automatic file format detection, no need to worry about file formats
  - Currently supported: mseed, sac, segy, seg2, gse1/2, seisan, sh, datamark, css, wav, y, Q, ...

– More file formats are included whenever a basic reading routine is provided (or e.g. sufficient documentation on data compression etc.)

– Custom user-specific file formats can be added (through plugin) to filetype autodiscovery in local ObsPy installation by user

Open a stream with wildcards

```
1 from obspy import read
2 st = read("./data/waveform_*")
3 print(st)
```

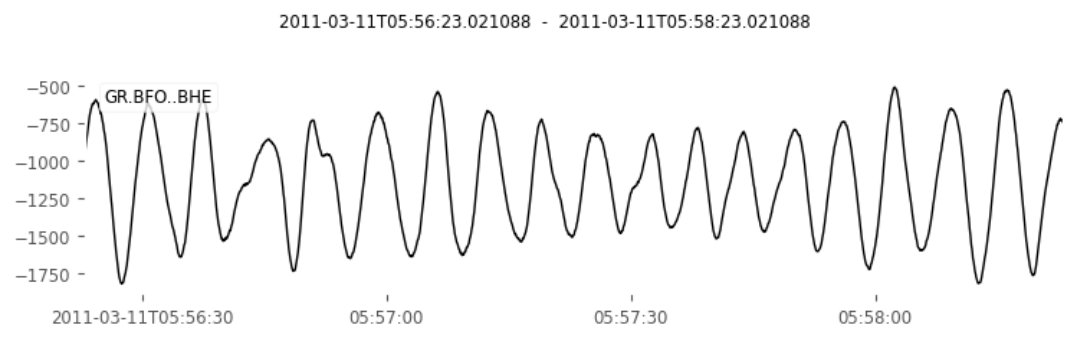
```
8 Trace(s) in Stream:
GR.BFO..BHE   | 2011-03-11T05:46:23.021088Z - 2011-03-11
   T06:36:22.971088Z | 20.0 Hz, 60000 samples
GR.BFO..BHN   | 2011-03-11T05:46:23.021088Z - 2011-03-11
   T06:36:22.971088Z | 20.0 Hz, 60000 samples
GR.BFO..BHZ   | 2011-03-11T05:46:23.021088Z - 2011-03-11
   T06:36:22.971088Z | 20.0 Hz, 60000 samples
II.PFO.00.BHZ | 2011-03-11T05:46:23.019500Z - 2011-03-11
   T06:36:22.969500Z | 20.0 Hz, 60000 samples
II.PFO.10.BHZ | 2011-03-11T05:46:23.019500Z - 2011-03-11
   T06:36:22.994500Z | 40.0 Hz, 120000 samples
SY.PFO.S3.MXE | 2011-03-11T05:47:31.587750Z - 2011-03-11
   T06:36:22.974250Z | 6.2 Hz, 18152 samples
SY.PFO.S3.MXN | 2011-03-11T05:47:31.587750Z - 2011-03-11
   T06:36:22.974250Z | 6.2 Hz, 18152 samples
SY.PFO.S3.MXZ | 2011-03-11T05:47:31.587750Z - 2011-03-11
   T06:36:22.974250Z | 6.2 Hz, 18152 samples
```

For MiniSEED files, only reading short portions of files without all of the file getting read into memory is supported (saves time and memory when working on large collections of big files).

Open a miniSEED stream with UTCDateTime

```
1 from obspy import UTCDateTime
2
3 t = UTCDateTime("2011-03-11T05:46:23.015400Z")
4 st = read("./data/waveform_*", starttime=t + 10 * 60,
     endtime=t + 12 * 60)
5 print(st)
6 st[0].plot()
```


```
8 Trace(s) in Stream:
GR.BFO..BHE   | 2011-03-11T05:56:23.021088Z - 2011-03-11
   T05:58:23.021088Z | 20.0 Hz, 2401 samples
GR.BFO..BHN   | 2011-03-11T05:56:23.021088Z - 2011-03-11
   T05:58:23.021088Z | 20.0 Hz, 2401 samples
GR.BFO..BHZ   | 2011-03-11T05:56:23.021088Z - 2011-03-11
   T05:58:23.021088Z | 20.0 Hz, 2401 samples
II.PFO.00.BHZ | 2011-03-11T05:56:23.019500Z - 2011-03-11
   T05:58:23.019500Z | 20.0 Hz, 2401 samples
II.PFO.10.BHZ | 2011-03-11T05:56:23.019500Z - 2011-03-11
   T05:58:23.019500Z | 40.0 Hz, 4801 samples
SY.PFO.S3.MXE | 2011-03-11T05:56:23.084250Z - 2011-03-11
   T05:58:23.078750Z | 6.2 Hz, 744 samples
SY.PFO.S3.MXN | 2011-03-11T05:56:23.084250Z - 2011-03-11
   T05:58:23.078750Z | 6.2 Hz, 744 samples
SY.PFO.S3.MXZ | 2011-03-11T05:56:23.084250Z - 2011-03-11
   T05:58:23.078750Z | 6.2 Hz, 744 samples
```



The Stream Object

When you create an instance of the `Stream` class, you can add inside one or several `Trace` object where the data and information about the stations are stored.

Stream object

```
1 from obspy import read
2 st = read("./data/waveform_PFO.mseed")
3 print(type(st))
4 print(st)
5 print(st.traces)
6 print(st[0])
```

```
<class 'obspy.core.stream.Stream'>

2 Trace(s) in Stream:
II.PFO.00.BHZ | 2011-03-11T05:46:23.019500Z - 2011-03-11
   T06:36:22.969500Z | 20.0 Hz, 60000 samples
II.PFO.10.BHZ | 2011-03-11T05:46:23.019500Z - 2011-03-11
   T06:36:22.994500Z | 40.0 Hz, 120000 samples

[<obspy.core.trace.Trace object at 0x7f86f0df8080>, <obspy
   .core.trace.Trace object at 0x7f86f0f64898>]

II.PFO.00.BHZ | 2011-03-11T05:46:23.019500Z - 2011-03-11
   T06:36:22.969500Z | 20.0 Hz, 60000 samples
```

The operator `+` is working with `Stream` objects and you can assemble more `Trace` together.

Assemble `Stream` objects

```
1  st1 = read("./data/waveform_PFO.mseed")
2  st2 = read("./data/waveform_PFO_synthetics.mseed")
3
4  st = st1 + st2
5  print(len(st), 'Traces')
6
7  st3 = read("./data/waveform_BFO_BHE.sac")
8
9  st += st3
10 print(len(st), 'Traces')
```

```
5 Traces
6 Traces
```

You can loop inside of the `Stream` object to access each `Trace` of it. A `Stream` is useful to apply the same process to a large number of waveforms or to group `Traces` for another process (for example with three components of one station in one `Stream`)

Loop into a `Stream` object

```
1 for tr in st:
2     print(tr.id)
```

```
II.PFO.00.BHZ
II.PFO.10.BHZ
SY.PFO.S3.MXE
SY.PFO.S3.MXN
SY.PFO.S3.MXZ
GR.BFO..BHE
```

The Trace Object

A `Trace` object is a single, contiguous waveform data block (i.e. regularly spaced time series, no gaps). It contains a limited amount of metadata in a dictionary-like object (`Trace.stats`) that fully describes the time series by specifying the recording location/instrument (network, station, location and channel code), the start time and the sampling rate.

For custom applications it is sometimes necessary to directly manipulate the metadata of a `Trace`. The metadata of the `Trace` will stay consistent, as all values are derived from the `starttime`, the `data` and the `sampling_rate` are updated automatically.

Manipulate `Trace` object with attributes

```
1 st = read("./data/waveform_PFO.mseed")
2 tr = st[0]  # get the first Trace in the Stream
3
4 print(tr.stats.delta, "|", tr.stats.endtime)
5 tr.stats.sampling_rate = 5.0
6 print(tr.stats.delta, "|", tr.stats.endtime)
7
```

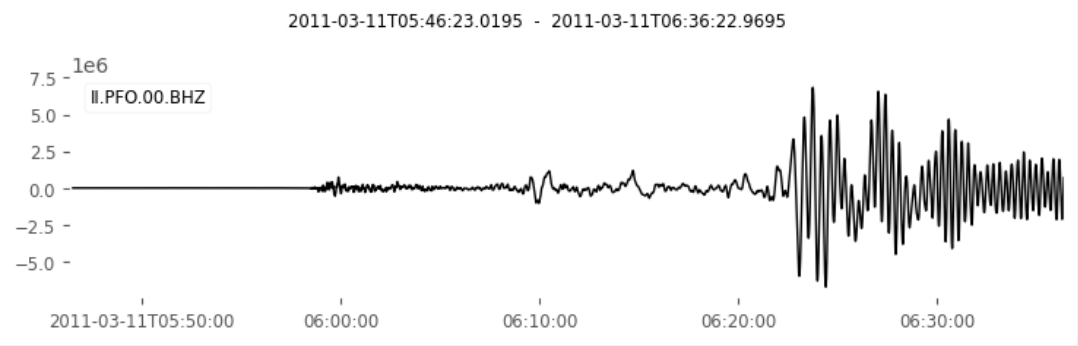
```
8  print(tr.stats.npts)
9  tr.data = tr.data[:100]
10 print(tr.stats.npts, "|", tr.stats.endtime)
```

```
0.05 | 2011-03-11T06:36:22.969500Z
0.2 | 2011-03-11T09:06:22.819500Z

60000 | 2011-03-11T06:36:22.969500Z
100 | 2011-03-11T05:46:42.819500Z
```

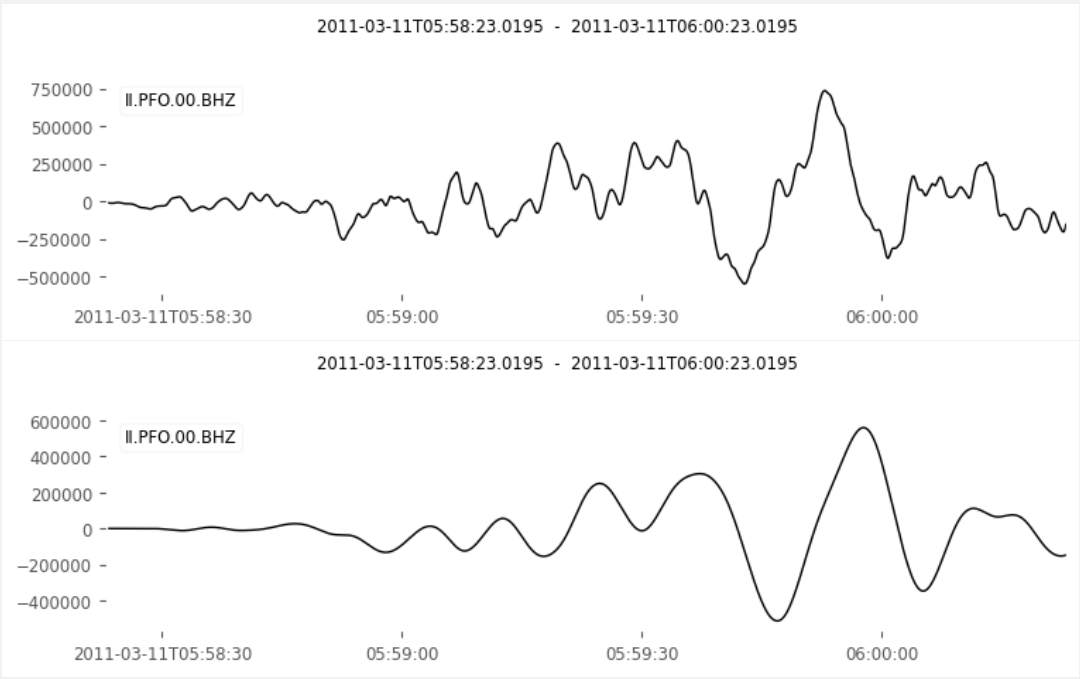
So it's possible to modify these attributes and to stay consistent between them but there are also methods included to do this job in a clean way.

Manipulate `Trace` object with methods

```
1  tr = read("./data/waveform_PFO.mseed")[0]
2  tr.plot()
3
4  print(tr)
5  tr.resample(sampling_rate=100.0)
6  print(tr)
7
8  print(tr)
9  tr.trim(tr.stats.starttime + 12 * 60, tr.stats.starttime
       + 14 * 60)
10 print(tr)
11 tr.plot()
12
13 tr.detrend("linear")
14 tr.taper(max_percentage=0.05, type='cosine')
15 tr.filter("lowpass", freq=0.1)
16 tr.plot()
```



```
II.PFO.00.BHZ | 2011-03-11T05:46:23.019500Z - 2011-03-11
   T06:36:22.969500Z | 20.0 Hz, 60000 samples
II.PFO.00.BHZ | 2011-03-11T05:46:23.019500Z - 2011-03-11
   T06:36:23.009500Z | 100.0 Hz, 300000 samples

II.PFO.00.BHZ | 2011-03-11T05:46:23.019500Z - 2011-03-11
   T06:36:23.009500Z | 100.0 Hz, 300000 samples
II.PFO.00.BHZ | 2011-03-11T05:58:23.019500Z - 2011-03-11
   T06:00:23.019500Z | 100.0 Hz, 12001 samples
```



The seismic data are directly available in the attribute `Trace.data` (so `tr.data` in our example). They are in the `ndarray` data type of the `numpy` library. Therefore, it's possible to apply directly a mathematical process on it. More information about this data type: http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html

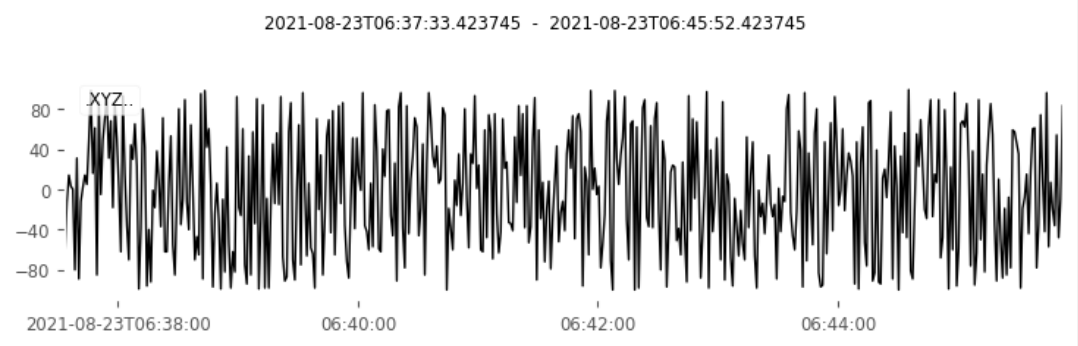
Possible operations on the seismic data

```
# operation on data
tr.data[:20]
tr.data ** 2 + 0.5

# builtin methods of ndarray
tr.data.max()
tr.data.mean()
tr.data.ptp()

# from the numpy library
import numpy as np
np.abs(tr.data)
```

Trace objects can also be manually generated with data in a `numpy.ndarray`

Generation of `Trace` data

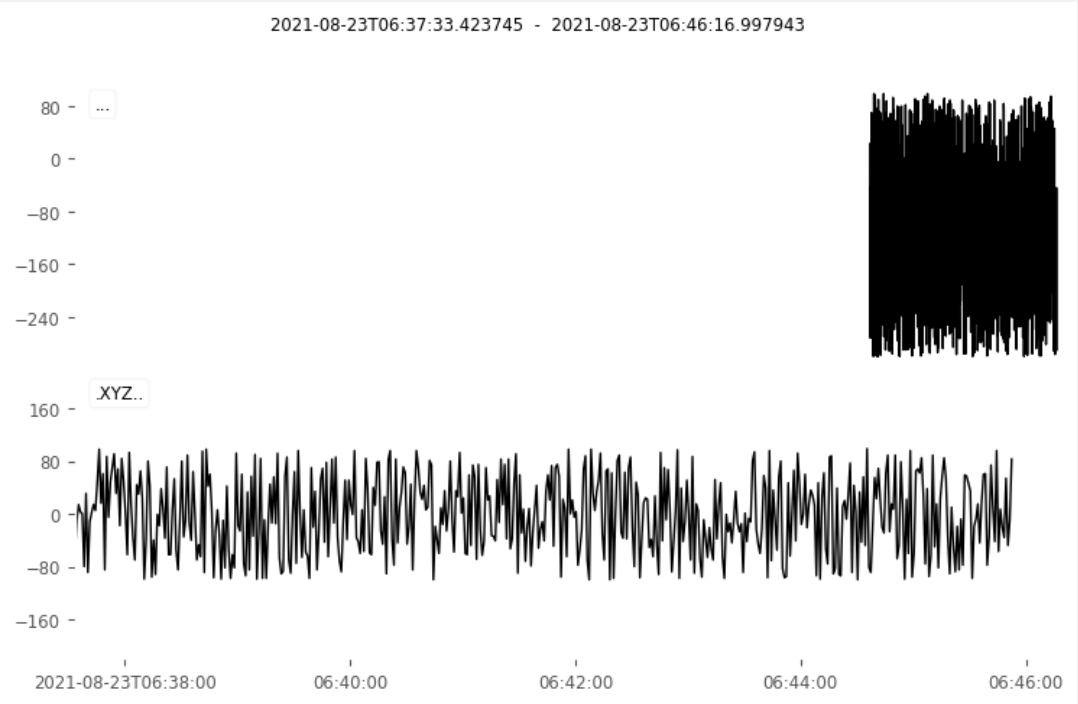
```
from obspy import Trace
import numpy as np

x = np.random.randint(-100, 100, 500)
tr = Trace(data=x)
tr.stats.station = "XYZ"
tr.stats.starttime = UTCDateTime()

tr.plot()
```



`Stream` objects can be assembled from manually generated `Trace` objects.

Generation of `Stream` object

```
1 from obspy import Stream
2
3 tr2 = Trace(data=np.random.randint(-300, 100, 1000))
4 tr2.stats.starttime = UTCDateTime()
5 tr2.stats.sampling_rate = 10.0
6 st = Stream([tr, tr2])
7
8 st.plot()
```



Builtin methods defined on `Stream` / `Trace`

Most methods that work on a Trace object also work on a Stream object. They are simply executed for every trace. See ObsPy documentation for an overview of available methods (http://docs.obspy.org/packages/autogen/obspy.core.stream.Stream.html):

- `st.filter()` - Filter all attached traces.
- `st.trim()` - Cut all traces.
- `st.resample()` / `st.decimate()` - Change the sampling rate.
- `st.trigger()` - Run triggering algorithms.
- `st.plot()` / `st.spectrogram()` - Visualize the data.

- `st.attach_response()` / `st.remove_response()`, `st.simulate()` - Instrument correction
- `st.merge()`, `st.normalize()`, `st.detrend()`, `st.taper()`, ...

A `Stream` object can also be exported to many formats, so ObsPy can be used to convert between different file formats.

5.2. Station and Event Metadata

5.2.1 Station Metadata

The standard format for Station Metadata is **FDSN StationXML**: http://www.fdsn.org/xml/station/. It is now used everywhere.

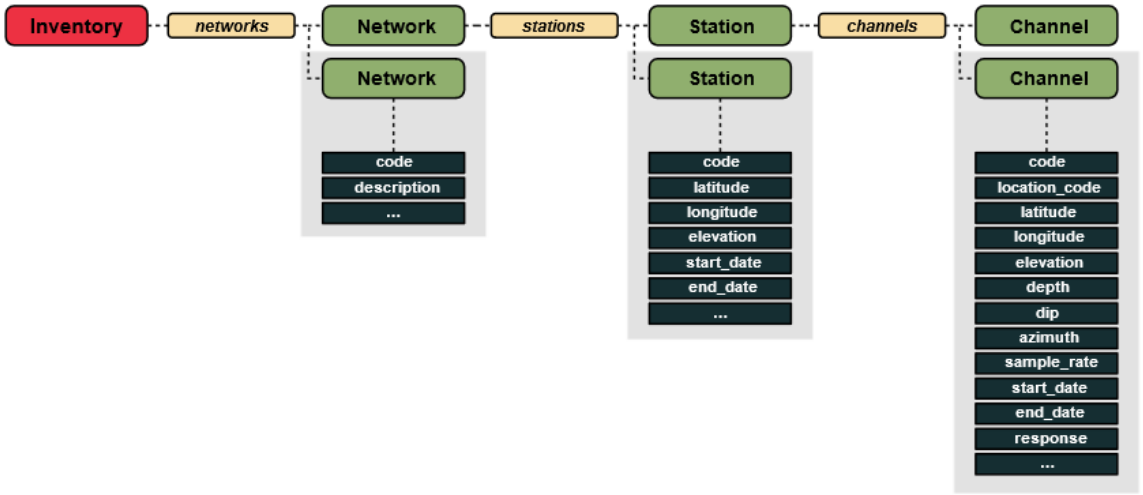
FDSN StationXML files can be read using `read_inventory()`.

Station metadata

```
1 from obspy import read_inventory
2
3 inventory = read_inventory("data/station_PFO.xml", format
    ="STATIONXML")
4 print(type(inventory))
5 print(inventory)
```

```
<class 'obspy.core.inventory.inventory.Inventory'>

Inventory created at 2014-02-17T15:43:45.000000Z
    Created by: IRIS WEB SERVICE: fdsnws-station | version
       : 1.0.11
            http://service.iris.edu/fdsnws/station/1/query
               ?network=II&level=res...
    Sending institution: IRIS-DMC (IRIS-DMC)
    Contains:
        Networks (1):
            II
        Stations (1):
            II.PFO (Pinon Flat, California, USA)
        Channels (2):
            II.PFO.00.BHZ, II.PFO.10.BHZ
```


The nested ObsPy Inventory class structure (Inventory/Station/Channel/Response/...) is closely modelled after FDSN StationXML.



You can access first network by network like a list, then the stations and finally the channels.

Access into the `inventory` file 1/3

```
1 network = inventory[0]
2 print(type(network))
3 print(network)
```

```
<class 'obspy.core.inventory.network.Network'>

Network II (Global Seismograph Network (GSN - IRIS/IDA))
    Station Count: 1/50 (Selected/Total)
    1986-01-01T00:00:00.000000Z - 2020-12-12T23
       :59:59.000000Z
    Access: open
    Contains:
        Stations (1):
            II.PFO (Pinon Flat, California, USA)
        Channels (2):
            II.PFO.00.BHZ, II.PFO.10.BHZ
```

Access into the inventory file 2/3

```
1 station = network[0]
2 print(type(station))
3 print(station)
```

```
<class 'obspy.core.inventory.station.Station'>

Station PFO (Pinon Flat, California, USA)
    Station Code: PFO
    Channel Count: 2/250 (Selected/Total)
    2006-07-13T00:00:00.000000Z - 2020-12-31T23
       :59:59.000000Z
    Access: open
    Latitude: 33.61, Longitude: -116.46, Elevation: 1280.0
        m
    Available Channels:
        PFO.00.BHZ, PFO.10.BHZ
```


Access into the `inventory` file 3/3

```
channel = station[0]
print(type(channel))
print(channel)
print(channel.response)
```

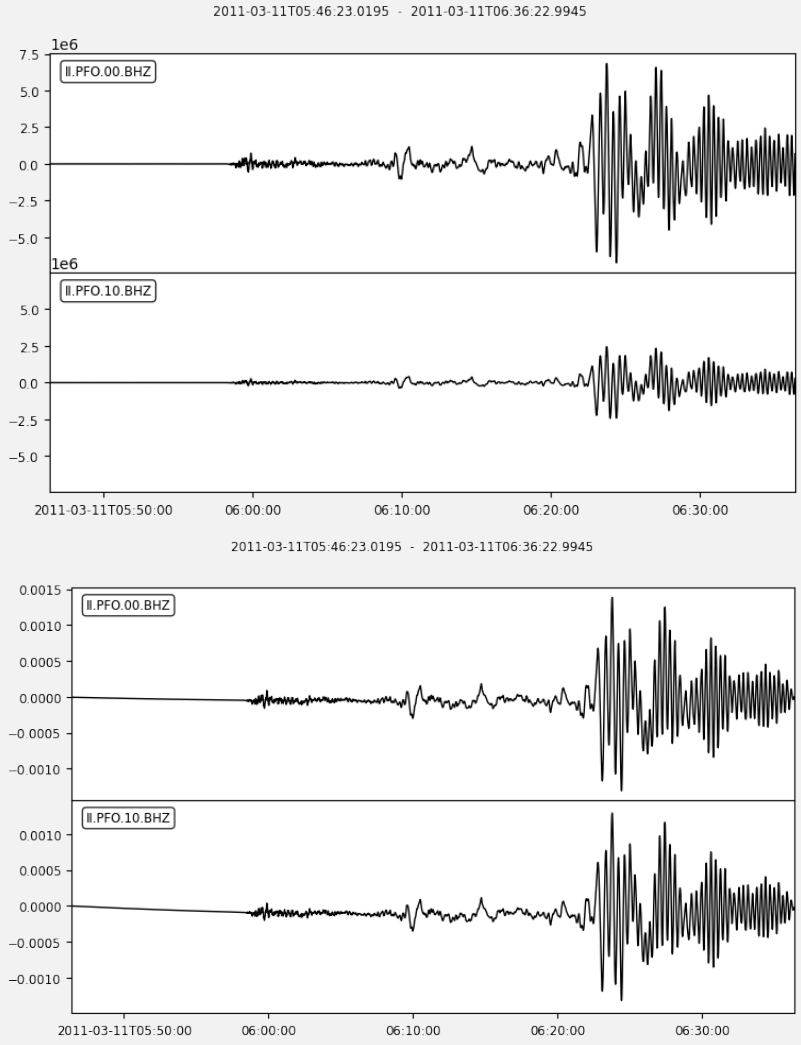
```
<class 'obspy.core.inventory.station.Station'>

Station PFO (Pinon Flat, California, USA)
    Station Code: PFO
    Channel Count: 2/250 (Selected/Total)
    2006-07-13T00:00:00.000000Z - 2020-12-31T23
       :59:59.000000Z
    Access: open
    Latitude: 33.61, Longitude: -116.46, Elevation: 1280.0
        m
    Available Channels:
        PFO.00.BHZ, PFO.10.BHZ

Channel Response
    From M/S (Velocity in Meters Per Second) to COUNTS (
       Digital Counts)
    Overall Sensitivity: 5.24814e+09 defined at 0.050 Hz
    4 stages:
        Stage 1: PolesZerosResponseStage from M/S to V,
           gain: 3314.4
        Stage 2: PolesZerosResponseStage from V to V, gain
           : 1
        Stage 3: CoefficientsTypeResponseStage from V to
           COUNTS, gain: 1.58333e+06
        Stage 4: CoefficientsTypeResponseStage from COUNTS
            to COUNTS, gain: 1
```

The instrument response can be removed from the waveform data using the convenience method `Stream.remove_response()` when you have the inventory file opened in your script.

Remove the instrument response

```
1 from obspy import read
2 st = read("data/waveform_PFO.mseed")
3 print(st)
4 inv = read_inventory("data/station_PFO.xml")
5 st.plot()
6 st.remove_response(inventory=inv)
7 st.plot()
```

```
2 Trace(s) in Stream:
II.PFO.00.BHZ | 2011-03-11T05:46:23.019500Z - 2011-03-11
   T06:36:22.969500Z | 20.0 Hz, 60000 samples
II.PFO.10.BHZ | 2011-03-11T05:46:23.019500Z - 2011-03-11
   T06:36:22.994500Z | 40.0 Hz, 120000 samples
```



5.2.2 Event Metadata

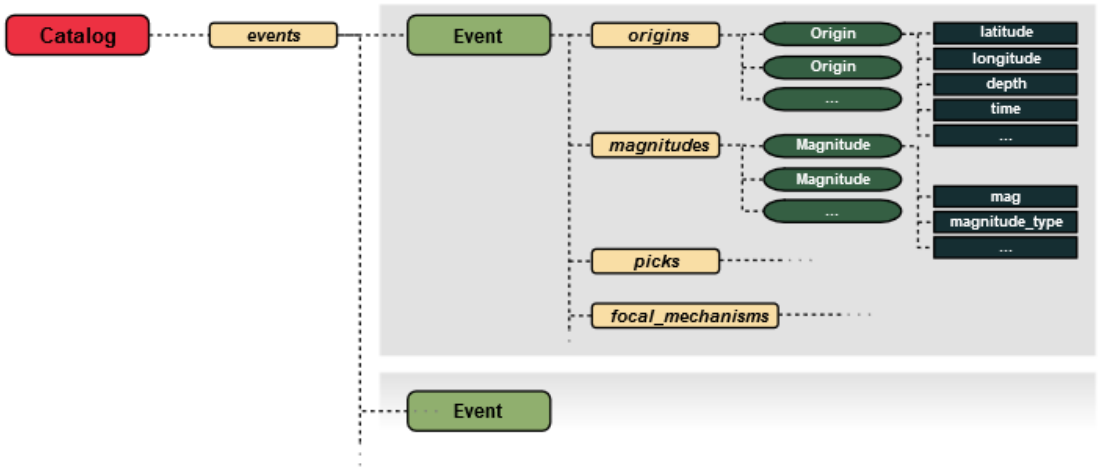
The common format is the **QuakeML**: https://quake.ethz.ch/quakeml/. To read it with the ObsPy library, you need to use the function `read_events()`. It returns a `Catalog` object, which is a collection of `Event` objects.

`Catalog` object for Event metadata

```
1 from obspy import read_events
2
3 catalog = read_events("data/
    event_tohoku_with_big_aftershocks.xml")
4 print(catalog)
5 print(type(catalog))
6 event = catalog[0]
7 print(type(event))
```

```
5 Event(s) in Catalog:
2011-03-11T05:46:23.200000Z | +38.296, +142.498 | 9.1 MW
2011-03-11T06:15:37.570000Z | +36.227, +141.088 | 7.9 MW
2011-03-11T06:25:50.740000Z | +38.051, +144.630 | 7.6 MW
2011-04-07T14:32:43.290000Z | +38.276, +141.588 | 7.1 MW
2011-07-10T00:57:10.800000Z | +38.034, +143.264 | 7.0 MW

<class 'obspy.core.event.catalog.Catalog'>
<class 'obspy.core.event.event.Event'>
```

The nested ObsPy Event class structure (Catalog/Event/Origin/Magnitude/FocalMechanism/...) is closely modelled after QuakeML file structure.



In the class `Event`, there's a lot of different other variables with all the information you need to study an seismic event.

Event object with Origin

```
print(type(event.origins))
print(type(event.origins[0]))
print(event.origins[0])
```

```
<class 'list'>
<class 'obspy.core.event.origin.Origin'>
Origin
        resource_id: ResourceIdentifier(id="smi:service.
           iris.edu/fdsnws/event/1/query?originid=9933375")
                time: UTCDateTime(2011, 3, 11, 5, 46, 23,
                   200000)
           longitude: 142.498
            latitude: 38.2963
               depth: 19700.0
      creation_info: CreationInfo(author='ISC')
```

Event object with Magnitude

```
print(type(event.magnitudes))
print(type(event.magnitudes[0]))
print(event.magnitudes[0])
```

```
<class 'list'>
<class 'obspy.core.event.magnitude.Magnitude'>
Magnitude
         resource_id: ResourceIdentifier(id="smi:service.
            iris.edu/fdsnws/event/1/query?magnitudeid
            =16642444")
                 mag: 9.1
      magnitude_type: 'MW'
           origin_id: ResourceIdentifier(id="smi:service.
            iris.edu/fdsnws/event/1/query?originid
            =9933383")
       creation_info: CreationInfo(author='GCMT')
```

When you use a Python software to write scripts, you can press `Tab` when writing `event.` to see all the attributes you have inside of the structure of an event.

To work with events, some methods are already implemented inside of the class to make things easier. For example, you can filter the events by magnitude.

Event object filtered by magnitude

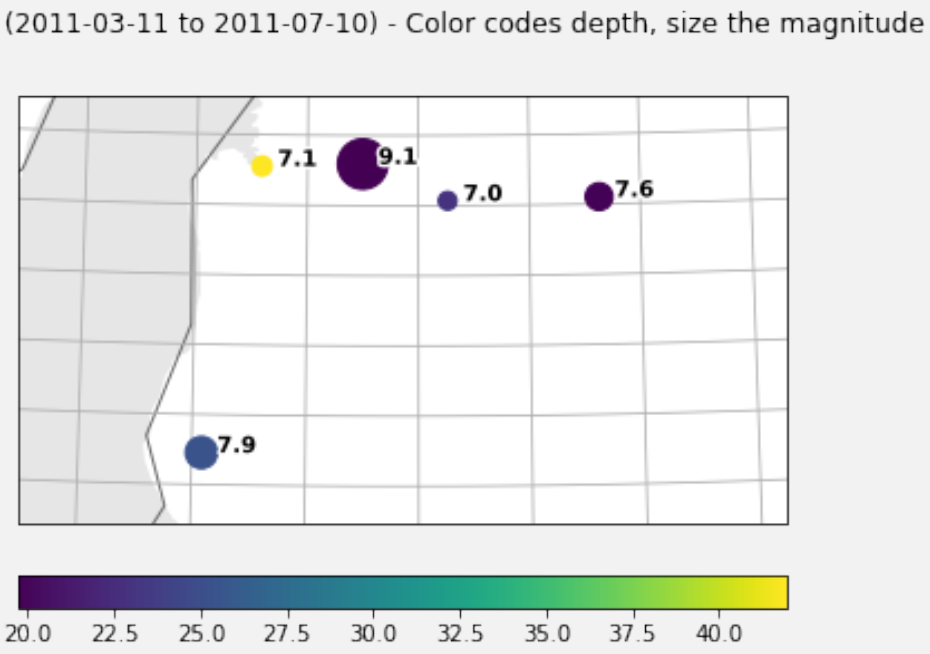
```
largest_magnitude_events = catalog.filter("magnitude >= 7.8")
print(largest_magnitude_events)
```

```
2 Event(s) in Catalog:
2011-03-11T05:46:23.200000Z | +38.296, +142.498 | 9.1 MW
2011-03-11T06:15:37.570000Z | +36.227, +141.088 | 7.9 MW
```

You can have preview with maps of the events with some libraries like Cartopy.

Plot map of an Event

```
catalog.plot(projection="local", outfile='catalog_plot.png')
```



You can save as a QuakeML file a modified `Catalog` with the method `write` of its class.

The `Event` type class can be used to build up `Event/Catalog/Picks/...` from scratch in custom processing work flows and to share them with other researchers in the standard format **QuakeML**.

Example of the creation of a `Catalog`

```
from obspy import UTCDateTime
from obspy.core.event import Catalog, Event, Origin,
    Magnitude
from obspy.geodetics import FlinnEngdahl

cat = Catalog()
cat.description = "Just an example catalog"

e = Event()
e.event_type = "not existing"

o = Origin()
o.time = UTCDateTime(2014, 2, 23, 18, 0, 0)
o.latitude = 47.6
o.longitude = 12.0
o.depth = 10000
o.depth_type = "operator assigned"
o.evaluation_mode = "manual"
o.evaluation_status = "preliminary"
o.region = FlinnEngdahl().get_region(o.longitude, o.
    latitude)

m = Magnitude()
m.mag = 7.2
m.magnitude_type = "Mw"

m2 = Magnitude()
m2.mag = 7.4
m2.magnitude_type = "Ms"

cat.append(e)
e.origins = [o]
e.magnitudes = [m, m2]
m.origin_id = o.resource_id
m2.origin_id = o.resource_id

print(cat)
cat.write("my_custom_events.xml", format="QUAKEML")
```

```
1 Event(s) in Catalog:
2014-02-23T18:00:00.000000Z | +47.600,  +12.000 | 7.2 Mw |
    manual
```


5.3. Retrieve Data from Datacenters

ObsPy has clients to directly fetch data via different services:

- FDSN webservices (IRIS, Geofon/GFZ, USGS, NCEDC, SeisComp3, ...)
- ArcLink (EIDA, ...)
- Earthworm
- SeedLink (near-realtime servers)
- NERIES/NERA/seismicportal.eu
- NEIC
- SeisHub (local seismological database)

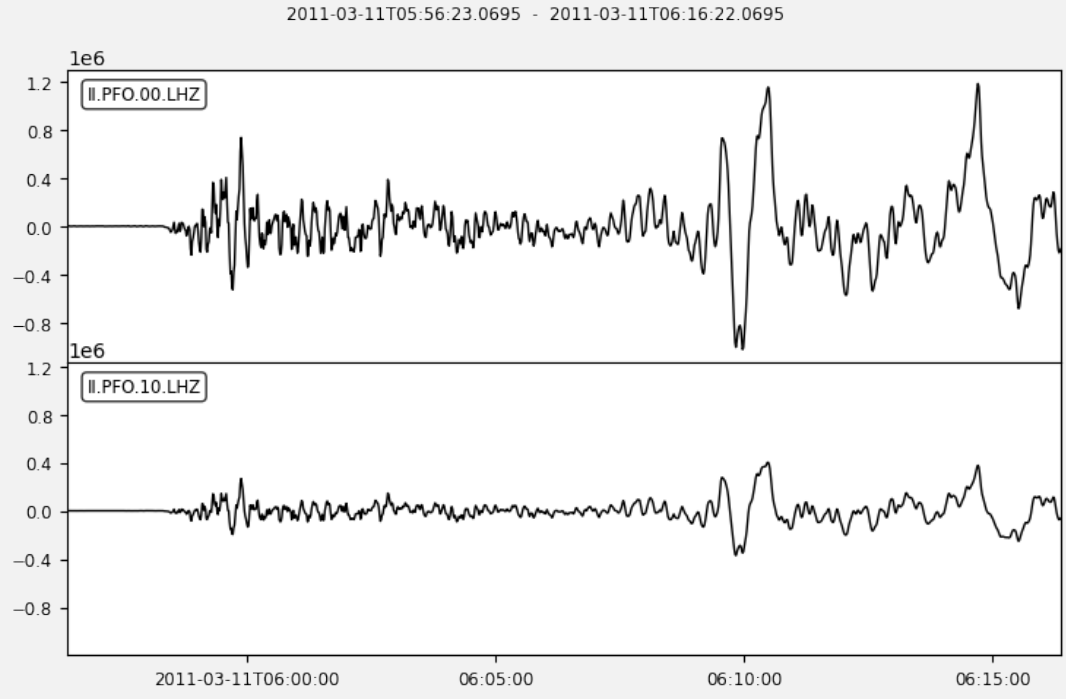
This introduction shows how to use the **FDSN webservice client**. The FDSN webservice definition is by now the default web service implemented by many data centers worldwide. Clients for other protocols work similar to the FDSN client.

5.3.1 Waveform Data

Waverform data with FDSN Client

```
1 from obspy import UTCDateTime
2 from obspy.clients.fdsn import Client
3
4 client = Client("IRIS")
5 t = UTCDateTime("2011-03-11T05:46:23")  # Tohoku
6 st = client.get_waveforms("II", "PFO", "*", "LHZ",
7                           t + 10 * 60, t + 30 * 60)
8 print(st)
9 st.plot()
```

```
2 Trace(s) in Stream:
II.PFO.00.LHZ | 2011-03-11T05:56:23.069500Z - 2011-03-11
   T06:16:22.069500Z | 1.0 Hz, 1200 samples
II.PFO.10.LHZ | 2011-03-11T05:56:23.069500Z - 2011-03-11
   T06:16:22.069500Z | 1.0 Hz, 1200 samples
```



For a Seedlink client, you have to import:

```
1 from obspy.clients.seedlink.basic_client import Client
```

The waveforms will be returned as a `Stream` and all the signal processing methods you can apply will work the same way as local file.

5.3.2 Event Metadata

The FDSN client can also be used to request event metadata.

Event metadata with FDSN Client

```
1 t = UTCDateTime("2011-03-11T05:46:23")  # Tohoku
2 catalog = client.get_events(starttime=t - 100, endtime=t
    + 24 * 3600, minmagnitude=9)
3 print(catalog)
```

```
1 Event(s) in Catalog:
2011-03-11T05:46:23.200000Z | +38.296, +142.498 | 9.1 MW
```

The requests have a wide range of different constraints (Time range, Geographical, Depth range, Magnitude range, Type, ...). To know more about it: http://docs.obspy.org/packages/autogen/obspy.fdsn.client.Client.get_events.html.

5.3.3 Station Metadata

Finally, the FDSN client can be used to request station metadata. Stations can be looked up, as Event metadata, using a wide range of constraints: http://docs.obspy.org/packages/autogen/obspy.fdsn.client.Client.get_stations.html.

Station metadata with FDSN Client

```
lon = 7.63
lat = 51.96

inventory = client.get_stations(longitude=lon, latitude=
    lat, maxradius=2.5, level="station")
print(inventory)
```

```
Inventory created at 2021-06-24T07:12:32.000000Z
    Created by: IRIS WEB SERVICE: fdsnws-station | version
        : 1.1.47
            http://service.iris.edu/fdsnws/station/1/query
                ?level=station&latitu...
    Sending institution: IRIS-DMC (IRIS-DMC)
    Contains:
        Networks (5):
            BE, GE, NL, SG, SY
        Stations (19):
            BE.MEM (Membach, Belgium)
            BE.RCHB (Rochefort, Belgium)
            BE.UCC (Uccle, Brussels, Belgium) (2x)
            GE.FLT1 (Temp GEOFON Station Flechtingen,
                Germany)
            GE.HLG (UKiel/GEOFON Station Helgoland,
                Germany) (2x)
            GE.IBBN (RUB/GEOFON Station Ibbenbueren,
                Germany)
            GE.WLF (GEOFON Station Walferdange, Luxembourg
                ) (2x)
            NL.HGN (HEIMANSGROEVE, NETHERLANDS)
            SG.MEMB (Membach, Belgium)
            SG.RCHS (Rochefort, Belgium)
            SY.FLT1 (FLT1 synthetic)
            SY.HGN (HGN synthetic)
            SY.HLG (HLG synthetic)
            SY.IBBN (IBBN synthetic)
            SY.RCHB (RCHB synthetic)
            SY.WLF (WLF synthetic)
```

The `level=...` parameter is used to specify the level of detail in the requested inventory:

- `"network"`: only return information on networks matching the criteria
- `"station"`: return information on all matching stations
- `"channel"`: return information on available channels in all stations networks matching the criteria
- `"response"`: include instrument response for all matching channels (large result data size!)

Station metadata at `station` level

```
1 inventory = client.get_stations(network="OE", station="
    DAVA", level="station")
2 print(inventory)
```

```
Inventory created at 2021-06-24T07:13:31.000000Z
    Created by: IRIS WEB SERVICE: fdsnws-station | version
        : 1.1.47
            http://service.iris.edu/fdsnws/station/1/query
                ?network=OE&station=D...
    Sending institution: IRIS-DMC (IRIS-DMC)
    Contains:
        Networks (1):
            OE
        Stations (1):
            OE.DAVA (Damuels, Vorarlberg, Austria)
        Channels (0):
```

Station metadata at `channel` level

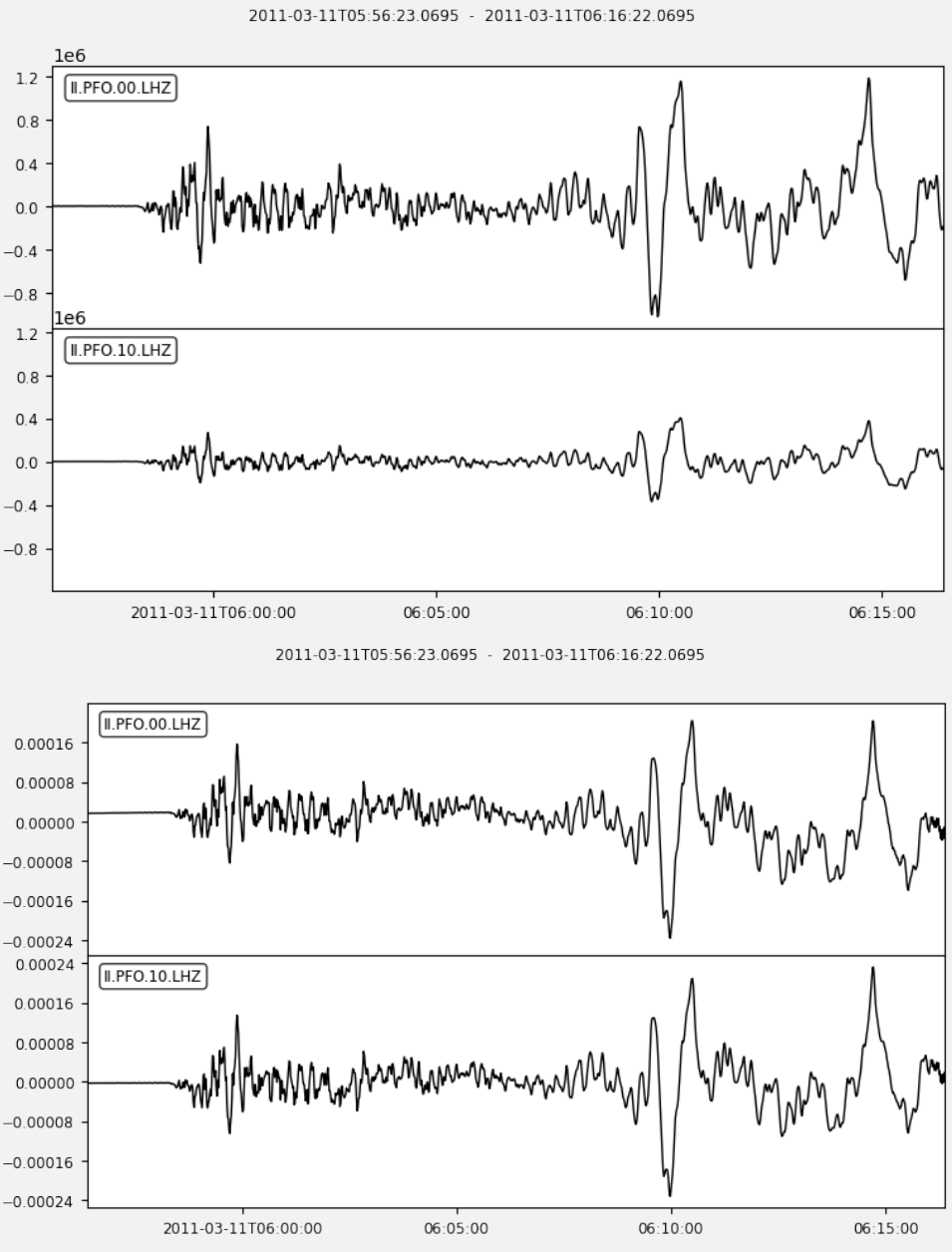
```
1 inventory = client.get_stations(network="OE", station="
    DAVA", level="channel")
2 print(inventory)
```

```
Inventory created at 2021-06-24T07:13:43.000000Z
    Created by: IRIS WEB SERVICE: fdsnws-station | version
       : 1.1.47
            http://service.iris.edu/fdsnws/station/1/query
               ?network=OE&station=D...
    Sending institution: IRIS-DMC (IRIS-DMC)
    Contains:
        Networks (1):
            OE
        Stations (1):
            OE.DAVA (Damuels, Vorarlberg, Austria)
        Channels (14):
            OE.DAVA..BHZ, OE.DAVA..BHN, OE.DAVA..BHE, OE.
               DAVA..HHZ,
            OE.DAVA..HHN, OE.DAVA..HHE, OE.DAVA..LCQ, OE.
               DAVA..LHZ,
            OE.DAVA..LHN, OE.DAVA..LHE, OE.DAVA..UFC, OE.
               DAVA..VHZ,
            OE.DAVA..VHN, OE.DAVA..VHE
```

For waveform requests that include instrument correction, the appropriate instrument response information can be attached to waveforms automatically (of course, for work on large datasets, the better choice is to download all station information and avoid the internal repeated webservice requests).

Station metadata with attached response

```
1 t = UTCDateTime("2011-03-11T05:46:23")  # Tohoku
2 st = client.get_waveforms("II", "PFO", "*", "LHZ", t + 10
      * 60, t + 30 * 60, attach_response=True)
3 st.plot()
4 st.remove_response()
5 st.plot()
```



5.4. Exercises

5.4.1 Timestamps

Exercise 1

Calculate the number of days passed between today and the Tohoku main shock (`2011-03-11T05:46:23.2`).

```
1 from obspy import UTCDateTime
2
3 nb_days = # ...
```

Exercise 2

Make a list of 10 `UTCDateTime` objects, starting today at 10:00 with a spacing of 90 minutes.

Exercise 3

Below is a list of strings with origin times of magnitude 8+ earthquakes between 2000 and 2013 (fetched from IRIS). Assemble a list of inter-event times in days. Use matplotlib to display a histogram.

```
1  import matplotlib.pyplot as plt
2
3  times = ["2001-06-23T20:33:09", "2003-09-25T19:50:07",
4           "2004-12-23T14:59:00", "2004-12-26T00:58:52",
5           "2005-03-28T16:09:35", "2006-06-01T18:57:02",
6           "2006-06-05T00:50:31", "2006-11-15T11:14:14",
7           "2007-01-13T04:23:23", "2007-04-01T20:39:56",
8           "2007-08-15T23:40:58", "2007-09-12T11:10:26",
9           "2009-09-29T17:48:11", "2010-02-27T06:34:13",
10          "2011-03-11T05:46:23", "2012-04-11T08:38:36",
11          "2012-04-11T10:43:10", "2013-05-24T05:44:48"]
```


5.4.2 Traces

The four parts constitutes a single exercise.

Part 1

1. Make an `numpy.ndarray` with zeros and (use `numpy.zeros()`) and put an ideal pulse somewhere in it
2. Initialize a Trace object with your data array
3. Fill in some station information (`network`, `station`, ...)
4. Print trace summary and plot the trace
5. Change the sampling rate to 20 Hz
6. Change the `starttime` of the `Trace` to the start time of this session
7. Print the `Trace` summary and plot the `Trace` again

```
1 import numpy as np
2 from obspy import Trace, UTCDateTime
3
4 # ...
```

Part 2

1. `Use tr.filter(...)` and apply a lowpass filter with a corner frequency of 1 Hz.
2. Display the plot, there are a few seconds of zeros that we can cut off.

Part 3

1. Use `tr.trim(...)` to remove some of the zeros at start and at the end
2. Show the plot again

Part 4

1. Scale up the amplitudes of the trace by a factor of 500
2. Add standard normal Gaussian noise to the `Trace` (use `np.random.randn()`)
3. Display the plot again

5.4.3 Stream

The four parts constitutes a single exercise. The data used are available on GitHub in the `data` folder: https://github.com/jeremow/Python_lessons/tree/main/data.

Part 1

1. Read all Tohoku example earthquake data into a `Stream` object (`./data/wave form_*`)
2. Print the stream summary

```
1 from obspy import read
2
3 # ...
```

Part 2

1. Use `st.select()` to only keep traces of station BFO in the `Stream`
2. Display the plot

Part 3

1. Trim the data to a 10 minute time window around the first arrival (just roughly looking at the previous plot)
2. Display the plot

```
1 from obspy import UTCDateTime
2
3 # ...
```

Part 4

1. Remove the linear trend from the data, apply a tapering and a lowpass at 0.1 Hertz
2. Display the plot

5.4.4 FDSN Client

Use the FDSN client to assemble a waveform dataset for an event. The four parts constitutes a single exercise.

Part 1

Search for a large earthquake on IRIS (by depth or in a region of your choice, use option `limit=5` to keep network traffic down).

```
1 from obspy.clients.fdsn import Client
2
3 # ...
```

Part 2

Search for stations to look at waveforms for the event. The stations should:

1. Be available at the time of the event
2. Have a vertical 1 Hz stream ("LHZ" for example)
3. Be in a narrow angular distance around the event (e.g. 90-91 degrees)
4. Adjust your search so that only a small number of stations (3-6) match your search criteria

Part 3

1. For each of these stations download data of the event, like a couple of minutes before to half an hour after the event
2. Put all data together in one `Stream` (put the `get_waveforms()` call in a `try/except` block to silently skip stations that actually have no data available)
3. Print stream info, plot the raw data

Part 4

Correct the instrument response for all stations with a `water_level=20` and plot the corrected data.

6. Correction of Exercises

6.1. Understanding of Python and its grammar

6.1.1 Flow Control Statement

Warmup 1

Print a text which is affected to a variable.

```
1 text = "Jeremy"
2 print(text)
```

```
Jeremy
```

Warmup 2

Transform your age into the good type to print it.

```
1 age = 26
2 text = "You're " + str(age) + " now!"
3 print(text)
```

```
You're 26 now!
```

Warmup 3

Ask a user his name and print it.

```
1 name = input("What's your name? ")
2 print("Your name is", name)
```

```
What's your name? Tuguldur
Your name is Tuguldur
```

Warmup 4

Create a converter that will ask for amount of days and convert it to years, then print it. Using `int()` function, convert the user's answer to integer. And then make your calculation.

```
1 nb_days = input("Number of days: ")
2 nb_days = int(nb_days)
3
4 nb_years = nb_days / 365
5 print(nb_days, "days =", nb_years, "years")
```

```
Number of days: 653
653 days = 1.789041095890411 years
```

Exercise 1

Print `Hello World` if `a` is greater than `b`.

```
1 a = 35
2 b = 25
3
4 if a > b:
5     print('Hello World')
```

```
Hello World
```

Exercise 2

Print `i` as long as `i` is less than 6.

First solution

```
1 i = 0
2 while i < 6:
3     print('i =', i)
4     i = i + 1 # or i += 1
```

```
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
```

Second solution

```
1 for i in range(0, 6):
2     print("i =", i)
```

```
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
```

Exercise 3

Loop/Browse through a string. Each time the letter *a* appears, print `Hey, I'm here`.

```
string = "SAin baina uu"

for letter in string:
    if letter == 'a' or letter == 'A':
        print("Hey, I'm here")
```

```
Hey, I'm here
Hey, I'm here
Hey, I'm here
```

Exercise 4

Given two integer numbers, print their product. If the product is greater than 1000, then print their sum.

```
a = 15
b = 100

product = a * b
print("Product:", product)

if product > 1000:
    sum_ab = a + b
    print('Sum:', sum_ab)
```

```
Product: 1500
Sum: 115
```

Exercise 5

Given a **range** of the first 10 numbers, Iterate from the start number to the end number, and In each iteration print the sum of the current number and previous number.

```
for i in range(1, 11):
    result = i + (i-1) # 2 * i - 1
    print(result)
```

```
1
3
5
7
9
11
13
15
17
19
```

Exercise 6

```
for i in range(1, 6):
    string = ''
    for j in range(1, i+1):
        string = string + ' ' + str(i)
    print(string)
```

```
 1
 2 2
 3 3 3
 4 4 4 4
 5 5 5 5 5
```

Exercise 7

Print each digit from an integer in the reverse order (For example 1234 gives 4321). Think about what does `number % 10`...

```
number = 1234
string = ""

while number != 0:
    string += str(number % 10)
    number = number // 10

print(string)
```

```
4321
```

Exercise 8

Calculate income tax for the given income by adhering to the below rules:

Taxable Income	Rate (in %)
First 500,000T	0
Between 500,000 and 1,000,000T	10
After 1,000,000T	20

```
income = 1500000

if income > 1000000:
    tax = (income - 1000000)*0.2 + 500000*0.1
elif income > 500000:
    tax = (income - 500000)*0.1
else:
    tax = 0

print("Taxes on {} income is {}".format(income, tax))
```

```
Taxes on 1500000 income is 150000.0
```

Exercise 9

Print multiplication table form 1 to 10.

```
for i in range(1, 11):
    line = str(i) + ": "
    for j in range(1, 11):
        line = line + str(i * j) + " "
    print(line)
```

```
1: 1 2 3 4 5 6 7 8 9 10
2: 2 4 6 8 10 12 14 16 18 20
3: 3 6 9 12 15 18 21 24 27 30
4: 4 8 12 16 20 24 28 32 36 40
5: 5 10 15 20 25 30 35 40 45 50
6: 6 12 18 24 30 36 42 48 54 60
7: 7 14 21 28 35 42 49 56 63 70
8: 8 16 24 32 40 48 56 64 72 80
9: 9 18 27 36 45 54 63 72 81 90
10: 10 20 30 40 50 60 70 80 90 100
```

Exercise 10

For a given word, add a star * after every letter of the word and print it at the end. Example: `word = 'python'` gives `p*y*t*h*o*n*`

```
word = "python"
result = ""

for letter in word:
    result = result + letter + "*"

print(result)
```

```
p*y*t*h*o*n*
```

Exercise 11

Count all letters, digits, and special symbols from a given string.

```
str1 = "P@#yn26at^&i5ve"
char = 0
digits = 0
symbol = 0

for element in str1:
    if element.isalpha():
        char += 1
    elif element.isnumeric():
        digits += 1
    else:
        symbol += 1

print("Chars = {}\nDigits = {}\nSymbol = {}".format(char,
    digits, symbol))
```

```
Chars = 8
Digits = 3
Symbol = 4
```

Exercise 12

Write a script that asks the user to enter their name and their age. Print out a message addressed to them that tells them the year that they will turn 100 years old.

```
name = input("What's your name: ")
age = int(input("How old are you: "))

print("In 100 years, {} will be {} years old".format(name
   , age+100))
```

```
What's your name: Jeremy
How old are you: 26
In 100 years, Jeremy will be 126 years old
```

Exercise 13

Ask the user for a number. Depending on whether the number is even or odd, print out an appropriate message to the user.

```
num = int(input("Write a number: "))
if num % 2 == 0:
    print("You wrote an even number")
else:
    print("You wrote an odd number")
```

```
Write a number: 3
You wrote an odd number
```

Exercise 14

Write a script which will print Fizz / Buzz or FizzBuzz from a user given input.

1. If the number is divisible by 3, it should print "Fizz".
2. If it is divisible by 5, it should print "Buzz".
3. If it is divisible by both 3 and 5, it should print "FizzBuzz".
4. Otherwise, it should print the number.

```
num = int(input("Write a number: "))

if num % 3 == 0 and num % 5 == 0:
    print("FizzBuzz")
elif num % 3 == 0:
    print("Fizz")
elif num % 5 == 0:
    print("Buzz")
else:
    print(num)
```

```
Write a number: 13
13
```

Exercise 15

Make a two-player Rock-Paper-Scissors game. Ask the two players their names. Then ask them what they want to play. After I let you think about it. And to finish, print the result and the winner.

Remember the rules:

1. Rock beats scissors
2. Scissors beats paper
3. Paper beats rock

```
1  user1 = input('Name of player 1: ')
2  user2 = input('Name of player 2: ')
3
4  print('For Rock, type 3, Scissors: 2, Paper: 1')
5
6  res_user1 = int(input('What do you want to play ' + user1
       +': '))
7  res_user2 = int(input('What do you want to play ' + user2
       +': '))
8
9  res = res_user1 - res_user2
10
11 if res == 1 or res == -2 :
12     print(user1, "wins against", user2)
13 elif res == 0:
14     print("It's a draw!")
15 else:
16     print(user2, "wins against", user1)
```

```
Name of player 1: Jeremy
Name of player 2: Tunga
For Rock, type 3, Scissors: 2, Paper: 1
What do you want to play Jeremy: 1
What do you want to play Tunga: 3
Jeremy wins against Tunga
```

6.1.2 Lists & Tuples

Warmup 1

Assign the first element of a list to a variable.

```
1 L = [1, 2, 3, 4]
2 a = L[0]
3 print(a)
```

```
1
```

Warmup 2

Insert an element to a specific index in a list.

```
1 L = [1, 2, 3, 4]
2 L.insert(2, "babuchka")
3 print(L)
```

```
[1, 2, 'babuchka', 3, 4]
```

Exercise 1

Write a script to sum all the items in a list using a for loop.

```
1 list_numbers = [1, 2, 3, 4]
2
3 answer = 0
4
5 for element in list_numbers:
6     answer += element
7
8 print(answer)
```

```
10
```


Exercise 2

Write a script to multiply all the numbers of a list with a constant.

Solution 1

```
1 list_numbers = [1, 2, 3, 4]
2 a = 10
3 list_numbers2 = []
4
5 for element in list_numbers:
6     list_numbers2.append(a * element)
7
8 print(list_numbers2)
```

```
[10, 20, 30, 40]
```

Solution 2

```
1 list_numbers = [1, 2, 3, 4, 15, 52]
2 a = 10
3
4 list_numbers2 = [a*element for element in list_numbers if
      element < 10]
5
6 print(list_numbers2)
```

```
[10, 20, 30, 40]
```

Exercise 3

Write a script to get the biggest number from a list using a loop and a condition inside. For you knowledge, there is the function `max` who return you directly the maximum, but the goal here is to manipulate lists ...

```
list_numbers = [1, 2, 304, 4, 15, 52]
max_nb = list_numbers[0]

for element in list_numbers[1:]:
    if element > max_nb:
        max_nb = element

print(max_nb)
```

```
304
```

Exercise 4

Write a script to copy a list. Be careful, you cannot just do `my_list1 = my_list2`.

```
my_list1 = [1, 2, 3, 4, 5]
my_list2 = []

for element in my_list1:
    my_list2.append(element)

print(my_list2)
```

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

Exercise 5

Write a script to append two lists together.

```
my_list1 = [1, 2, 3, 4, 5]
my_list2 = [6, 7, 8, 9, 10]

my_list1 += my_list2
print(my_list1)
```

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

Exercise 6

Write a script to sum all the items in list given by a user. You have to use `input` but you cannot just ask once. You have to make a while-loop condition or a for-loop with a precise number of elements you want to put in ... and append in your list each time for you pass in the loop. Think about it.

```
1  L = []
2  sum = 0
3  nb_element = int(input("How many elements in the list: ")
       )
4
5  for i in range(0, nb_element):
6      user_nb = int(input("Element number " + str(i) + ": "
           ))
7      sum += user_nb
8      L.append(user_nb)
9
10 print(L)
11 print(sum)
```

```
How many elements in the list: 3
Element number 0: 2
Element number 1: 5
Element number 2: 6
[2, 5, 6]
13
```

Exercise 7

Write a script to get the last two elements of a list.

```
1  L = [1, 2, 304, 4, 15, 52]
2
3  print(L[-2:])
```

```
[15, 52]
```

Exercise 8

Write a script to get a list with only the values of even indexes. $[15, 13, 59, 40]$ would give $[15, 59]$

```
1 L = [15, 13, 59, 40]
2 print(L[::2])
```

```
[15, 59]
```

Exercise 9

Write a script asking the user an index and then split the list into two different lists. $[1, 2, 3, 4]$ with the index 1 would give $[1, 2]$ and $[3, 4]$

```
1  L = [1, 2, 3, 4]
2
3  split_nb = int(input("Give the index to split: "))
4
5  if split_nb >= len(L):
6      split_nb = len(L) - 1
7
8  L1 = L[:split_nb + 1]
9  L2 = L[split_nb + 1:]
10 print(L1, L2)
```

```
Give the index to split: 6
[1, 2, 3, 4] []
```

Exercise 10

Write a script to insert a given string at the beginning of all items in a list.

Solution 1

```
1 list1 = [1, 2, 3, 4]
2 list2 = []
3 s = 'mongolia'
4
5 for i, element in enumerate(list1):
6     list2.append(s + str(list1[i]))
7
8 print(list2)
```

```
['mongolia1', 'mongolia2', 'mongolia3', 'mongolia4']
```

Solution 2

```
1  list1 = [1, 2, 3, 4]
2  list2 = []
3  s = 'mongolia'
4
5  for i in range(0, len(list1)):
6      list2.append(s + str(list1[i]))
7      list1[i] = s + str(list1[i])
8
9  print(list1)
10 print(list2)
```

```
['mongolia1', 'mongolia2', 'mongolia3', 'mongolia4']
['mongolia1', 'mongolia2', 'mongolia3', 'mongolia4']
```

Exercise 11

Write a script displaying all the elements of the list that are less than 5.

```
1 a = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
2 L = [element for element in a if element < 5]
3 print(L)
```

```
[1, 1, 2, 3]
```

Exercise 12

Write a script to create a 3X3 grid with numbers (Lists inside of a list).

```
L_model = [1, 2, 3]
L = []

for i in range(0, 3):
    L.append(L_model)

print(L)
```

```
[[1, 2, 3], [1, 2, 3], [1, 2, 3]]
```

Exercise 13

Write a script to sort a given list of numbers numerically (without the method `sort` of course).

```
L = [5, 1, 95, 43, 2]

for i in range(0, len(L)):
    for j in range(0, len(L)):
        if i == j:
            continue
        else:
            if L[i] < L[j]:
                L[i], L[j] = L[j], L[i]

print(L)
```

```
[1, 2, 5, 43, 95]
```

Exercise 14

Let's say I give you a list saved in a variable: $a = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]$. Write one line of Python that takes this list a and makes a new list that has only the even elements of this list in it. There is something you can do: it's using a for loop in a list structure in one line ...

```
a = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

list2 = [element for element in a if element % 2 == 0]

print(list2)
```

```
[4, 16, 36, 64, 100]
```

Exercise 15

Write a script that takes a list and print a new list that contains all the elements of the first list minus all the duplicates. (`if *** in ***`)

```
a = [1, 4, 1, 16, 1, 36, 49, 64, 1, 100]
b = []

for element in a:
    if element in b:
        continue
    else:
        b.append(element)
print(b)
```

```
[1, 4, 16, 36, 49, 64, 100]
```

6.1.3 Dictionaries

Warmup 1

Remove the element of the key `key4`.

```
1 my_dict = {'key1': 'hello', 'key2': 13, 'key3': [1, 2,
    3], 'key4': True}
2 my_dict.pop('key4')
3 print(my_dict)
```

```
{'key1': 'hello', 'key2': 13, 'key3': [1, 2, 3]}
```

Warmup 2

Add the boolean `False` to the key `key4`.

```
1 my_dict = {'key1': 'hello', 'key2': 13, 'key3': [1, 2,
    3]}
2 my_dict['key4'] = False
3 print(my_dict)
```

```
{'key1': 'hello', 'key2': 13, 'key3': [1, 2, 3], 'key4':
   False}
```

Warmup 3

Loop into the dictionary `fruits` and print the keys of it.

```
1 fruits = {'watermelon': 15, 'apple': 10, 'peach': 25, '
    orange': 30}
2 for key in fruits.keys():
3     print(key)
```

```
watermelon
apple
peach
orange
```

Warmup 4

Loop into the dictionary `fruits` and print the values of it.

```
1 fruits = {'watermelon': 15, 'apple': 10, 'peach': 25, '
    orange': 30}
2 for value in fruits.values():
3     print(value)
```

```
15
10
25
30
```

Warmup 5

Loop into the dictionary `fruits` and print the keys and values of it.

```
1 fruits = {'watermelon': 15, 'apple': 10, 'peach': 25, '
    orange': 30}
2 for key, value in fruits.items():
3     print(key, ':', value)
```

```
watermelon : 15
apple : 10
peach : 25
orange : 30
```

Exercise 1

Filter a dictionary based on values: you have a dictionary with name as keys and ages, and you want to create another dictionary with only the people under 18.

```
people = {'Jeremy': 26, 'Zayaa': 4, 'Zul': 16, 'Khulan':
   33, 'Tuguldur': 12, 'Buyanaa': 48}
children = {}

for key, value in people.items() :
    if value < 18:
        children[key] = value

print(children)
```

```
{'Zayaa': 4, 'Zul': 16, 'Tuguldur': 12}
```

Exercise 2

Find the highest value in the dictionary.

```
people = {'Jeremy': 26, 'Zayaa': 4, 'Zul': 16, 'Khulan':
   33, 'Tuguldur': 12, 'Buyanaa': 48}

max_value = 0
for key, value in people.items() :
    if value > max_value:
        max_value = value
        name = key

print('Highest value is', max_value,'for', name)
```

```
Highest value is 48 for Buyanaa
```

Exercise 3

Get the highest price in a shop and put it in tuple.

```
shop, max_item = {'apple': 6, 'pear': 12, 'coconut': 42,
    'mango': 33, 'ananas': 15}, ('',0)

for item in shop.items():
    if item[1] > max_item[1]:
        max_item = item

print(max_item)
```

```
('coconut', 42)
```

Exercise 4

Filter a dictionary based on values. For example for values over 30:

Solution 1

```
shop = {'apple': 6, 'pear': 12, 'coconut': 42, 'mango':
    33, 'ananas': 15}
result = {}

for key, value in shop.items():
    if item[1] > 30:
        result[key] = value

print(result)
```

Solution 2

```
shop = {'apple': 6, 'pear': 12, 'coconut': 42, 'mango':
    33, 'ananas': 15}
result = {key: value for key, value in shop.items() if
    value > 30}
print(result)
```

```
{'coconut': 42, 'mango': 33}
```

Exercise 5

Write a script to create a dictionary from a string. Be careful: You have to count when a letter appears several times.

```
my_string = 'erdenezul'
dict_str = {}

for letter in my_string:
    if letter in dict_str.keys():
        dict_str[letter] += 1
    else:
        dict_str[letter] = 1

print(dict_str)
```

```
{'e': 3, 'r': 1, 'd': 1, 'n': 1, 'z': 1, 'u': 1, 'l': 1}
```

Exercise 6

Date Decoder. A date of the form 8-MAR-2021 includes the name of the month, which must be translated to a number. Create a dictionary suitable for decoding month names to numbers. Use string operations to split the date into 3 items using the "-" character. It will accept a date in the `"dd-MMM-yyyy"` format and respond with a list of `[y, m, d]`.

```
date = "8-MAR-2021"
dict_month = {"JAN": 1, "FEB": 2, "MAR": 3, "APR": 4, "MAY
   ": 5, "JUN": 6, "JUL": 7, "AUG": 8, "SEP": 9, "OCT":
   10, "NOV": 11, "DEC": 12}

split_date = date.split("-")

day = int(split_date[0])
month = dict_month[split_date[1]]
year = int(split_date[2])

date_list = [year, month, day]
print(date_list)
```

```
[2021, 3, 8]
```

Exercise 7

You have a list of students in a list. It prints `True` if there is the value of the specified key exist. Else `False`.

```
students = [
    {'student_id': 1, 'name': 'Jeremy', 'class': '1'},
    {'student_id': 2, 'name': 'Tuguldur', 'class': '3'},
    {'student_id': 3, 'name': 'Pujee', 'class': '2'},
    {'student_id': 4, 'name': 'Demberel', 'class': '1'} ]

key, value = 'name', 'Demberel'
ans = False
for student in students:
    if student[key] == value:
        ans = True
print(ans)
```

```
True
```

Exercise 8

We will keep track of our friends' birthdays, and be able to find that information based on their name. Create a dictionary of names and birthdays. When you run your script it should ask the user to enter a name, and return the birthday of that person.

```
birth_dict = {'Albert Einstein': '03/14/1879', 'Benjamin
   Franklin': '01/17/1706', 'Ada Lovelace': '12/10/1815'}
print('Welcome. We know the birthdays of:')
for name in birth_dict.keys():
    print(name)
user_ans = input("Who's birthday do you want to look up?")
print("{}'s birthday is {}".format(user_ans, birth_dict[
   user_ans]))
```

```
Welcome. We know the birthdays of:
Albert Einstein
Benjamin Franklin
Ada Lovelace
Who's birthday do you want to look up? Albert Einstein
Albert Einstein's birthday is 03/14/1879
```

6.2. Functions and Modules

6.2.1 Functions

Warmup 1

Create a function `say_hello` which displays the message *"ooooo sain baina uu?"*

```
1 def say_hello():
2     print("ooooo sain baina uu?")
3
4 say_hello()
```

```
ooooo sain baina uu?
```

Warmup 2

Create a function `say_hello` which takes one parameter `name` and display *'`name`, sain baina uu?"*

```
1 def say_hello(name):
2     print('{}, sain baina uu?'.format(name))
3
4 say_hello('Baigalaa')
```

```
Baigalaa, sain baina uu?
```


Warmup 3

Create a function `present_user` which takes 2 parameters `name` and `age` and displays *"Minii ner `name`. Bi `age`-n nastai"*

```
1 def present_user(name, age):
2     print("Minii ner {}. Bi {}-n nastai".format(name, age
      ))
3
4 present_user('Jeremy', 26)
```

```
Minii ner Jeremy. Bi 26-n nastai
```

Warmup 4

Create a function `say_hello` which takes one parameter `name` and display *'`name`, sain baina uu?"*. Default value of `name` if not precised is *Zochin*.

```
1 def say_hello(name='Zochin'):
2     print('{}, sain baina uu?'.format(name))
3
4 say_hello('Tunga')
5 say_hello()
```

```
Tunga, sain baina uu?
Zochin, sain baina uu?
```

Warmup 5

Create a function `get_2x_age` which takes `age` as parameter, displays *"You're* `age` *and twice your age is :* `age_2x`*"* and return two times your age. Default value is *18*.

```
1  def get_2x_age(age=18):
2      twice_my_age = age * 2
3      print("You're {} and twice your age is : {}".format(
           age, twice_my_age))
4      return twice_my_age
5
6  twice_my_age = get_2x_age(26)
7  print(twice_my_age)
8
9  twice_my_age = get_2x_age()
10 print(twice_my_age)
```

```
You're 26 and twice your age is : 52
52
You're 18 and twice your age is : 36
36
```

Exercise 1

Create a function `above_limit` which takes in parameter a list `L` and an integer `limit` and **return** a new list with only the numbers of `L` which are above or equal to `limit`.

Solution 1

```
1  def above_limit(my_list, limit):
2      above_list = []
3      for element in my_list:
4          if element > limit:
5              above_list.append(element)
6      return above_list
7
8  my_list = [2, 15, 18, 25, 96]
9  above_list = above_limit(my_list, 16)
10 print(above_list) # must display [18, 25, 96]
```

Solution 2

```
1 def above_limit(my_list, limit):
2     above_list = [element for element in my_list if
          element > limit]
3     return above_list
4
5 my_list = [2, 15, 18, 25, 96]
6 above_list = above_limit(my_list, 16)
7 print(above_list) # must display [18, 25, 96]
```

```
[18, 25, 96]
```

Exercise 2

Create a function `first_elements` which takes a list `L` and an integer `nb_elements` in parameters and **return** a new list with only the first `nb_elements` elements of the list `L`. If `nb_elements` is superior to the length of the list `L`, return a new list with all the elements of `L` (don't return `L` itself!). To get the length of a list: `len(my_list)`.

```
1 def first_elements(L, nb_elements):
2     return L[:nb_elements]
3
4 my_list = [2, 15, 18, 25, 96]
5 my_new_list = first_elements(my_list, 2)
6 print(my_new_list) # must display [2, 15]
7
8 my_new_list = first_elements(my_list, 10)
9 print(my_new_list) # must display [2, 15, 18, 25, 96]
```

```
[2, 15]
[2, 15, 18, 25, 96]
```

Exercise 3

Create a function `last_elements` which takes a list `L` and an integer `nb_elements` in parameters and **return** a new list with only the last `nb_elements` elements of the list `L` beginning from the end. If `nb_elements` is superior to the length of the list `L`, return an empty list. To get the length of a list: `len(my_list)`.

Solution 1

```
1  def last_elements(L, nb_elements):
2      my_new_list = []
3      L.reverse()
4
5      if nb_elements > len(L):
6          return my_new_list
7
8      for element in L[0:nb_elements]:
9          my_new_list.append(element)
10
11     return my_new_list
12
13 my_list = [2, 15, 18, 25, 96]
14 my_new_list = last_elements(my_list, 2)
15 print(my_new_list) # must display [96, 25]
16
17 my_new_list = last_elements(my_list, 10)
18 print(my_new_list) # must display []
```

Solution 2

```
1  def last_elements(L, nb_elements):
2      if nb_elements > len(L):
3          return []
4      else:
5          nb_elements = len(L) - nb_elements
6          new_list = L[nb_elements:len(L)]
7          new_list.reverse()
8          return new_list
9
10 my_list = [2, 15, 18, 25, 96]
11 my_new_list = last_elements(my_list, 2)
12 print(my_new_list) # must display [96, 25]
13
14 my_new_list = last_elements(my_list, 10)
15 print(my_new_list) # must display []
```

```
[96, 25, 18]
[]
```


Exercise 4

Create a function `fact` which takes a number `n` as parameter and **return** the factorial of this number. It must be a non-negative integer. If `n` negative, display *"Number must be positive"* and return `None`. (Be careful, `fact(0) = 1` ...)

Solution 1: Basic mindset

```
1 def fact(n):
2     if n < 0:
3         print("Number must be positive")
4         return None
5
6     res = 1
7     for i in range(1, n+1):
8         res = res * i
9     return res
```

Solution 2: Recursive mindset

```
1 def fact(n):
2     if n < 0:
3         print("Number must be positive")
4         return None
5
6     if n == 0 or n == 1:
7         return 1
8     else:
9         return n * fact(n-1)
10
11 nb = fact(5)
12 print(nb) # must display 120
13
14 nb = fact(-1) # must display "Number must be positive"
15 print(nb) # must display None
```

```
120
Number must be positive
None
```

Exercise 5

Create a function `case_letters` that takes a string `string` as parameter and **displays** the number of upper case letters and lower case letters. The method to see if a character is in lower case: `char.islower()` return `True` if `char` is in lower case. For upper case: `char.isupper()` ...

```
def case_letters(string):
    lower_nb = 0
    upper_nb = 0
    for letter in string:
        if letter.isupper():
            upper_nb += 1
        elif letter.islower():
            lower_nb += 1
    print("Number of lower case letters: {} ; Number of
       upper case letters: {}.".format(lower_nb, upper_nb
       ))

case_letters('JeReMY') # must display: "Number of lower
   case letters: 2 ; Number of upper case letters: 4."
case_letters('Hello, I am a real Mongolian Woman')
# must display: "Number of lower case letters: 23 ;
   Number of upper case letters: 4."
```

```
Number of lower case letters: 2 ; Number of upper case
   letters: 4.
Number of lower case letters: 23 ; Number of upper case
   letters: 4.
```

Exercise 6

Create a function `is_pangram` which takes a string as parameter and **displays** whether the string is a pangram or not and **return** `True` or `False`. A pangram is a sentence with every letter of the alphabet at least once. To do it, you have to check every letter of the sentence and add to a defined dictionary the number of times you see a letter.

```
1  def is_pangram(string):
2      alphabet = {'a': 0, 'b': 0, 'c': 0, 'd': 0, 'e': 0, '
          f': 0, 'g': 0, 'h': 0, 'i': 0, 'j': 0, 'k': 0, 'l'
          : 0, 'm': 0, 'n': 0, 'o': 0, 'p': 0, 'q': 0, 'r':
          0, 's': 0, 't': 0, 'u': 0, 'v': 0, 'w': 0, 'x': 0,
           'y': 0, 'z': 0}
3
4      for letter in string:
5          if letter.isalpha():
6              alphabet[letter.lower()] += 1
7
8      for value in alphabet.values():
9          if value < 1:
10             print("The sentence isn't a pangram")
11             return False
12
13     print("The sentence is a pangram")
14     return True
15
16 answer = is_pangram('Portez ce vieux whisky au juge blond
       qui fume') # must display: "The sentence is a pangram
      "
17 print(answer) # must display True
18
19 answer = is_pangram('Hello, I am a real Mongolian Woman')
       # must display: "The sentence isn't a pangram"
20 print(answer) # must display False
```

```
The sentence is a pangram
True
The sentence isn't a pangram
False
```

Exercise 7

Write a little game **More Or Less** as a function `more_or_less` that randomly generates a number between two given numbers and ask the user to give a number until he finds it. You will write the function `more_or_less` which takes two parameters: the limits of the game. You'll also write the function `compare_number` which takes two numbers as parameters (the one which is generated by the game and the input one of the user) and displays if the number to guess is higher or lower than the user number and return `False`, displays *"You win"* if you find the number and return `True`.

```
1  import random
2
3  def compare_numbers(nb_to_guess, user_nb):
4      if user_nb < nb_to_guess:
5          print('You enter {}. The number to guess is
               higher'.format(user_nb))
6          return False
7      elif user_nb > nb_to_guess:
8          print('You enter {}. The number to guess is lower
               '.format(user_nb))
9          return False
10     else:
11         print('You enter the good number !')
12         return True
13
14 def more_or_less(low_lim, high_lim):
15
16     nb_to_guess = random.randint(low_lim, high_lim)
17     user_win = False
18
19     while user_win is not True:
20         # input number and convert into int ...
21         user_nb = int(input("Enter the number you think :
               "))
22         user_win = compare_numbers(nb_to_guess, user_nb)
23
24 more_or_less(1, 4)
```

```
Enter the number you think : 2
You enter 2. The number to guess is higher
Enter the number you think : 3
You enter 3. The number to guess is higher
Enter the number you think : 4
You enter the good number !
```

Exercise 8

The `print` function: Try below to rewrite the print function. Some indications : the function doesn't return anything, you have to convert all the arguments in the string format before sending your final string to the stream. The stream is where you display the printed object you want to see or users to see.

```
1  import sys
2
3  def your_print(*parameters, sep = ' ', end = '\n'):
4      """
5      Here is your print function. The parameters are in a
          tuple, the sep character is a space and the ending
6      character is backspace.
7      """
8      # get all the parameters one by one, convert them to
          str and add them the a final string called
          final_string
9      str_parameters = ''
10     final_string= ''
11     for _, parameter in enumerate(parameters):
12         str_parameters += str(parameter)
13         str_parameters += sep
14
15     final_string = str_parameters + end
16
17     sys.stdout.write(final_string)
18
19 your_print('Hello', 'World!')
20 your_print('Hello you')
```

```
Hello World!
Hello you
```

Exercise 9

You manage a fruit shop. Your wholesaler has a list of fruits available in infinite amount you can buy. His list will be composed of orange, apple, grape, watermelon, peach, banana, apricots, cherries. You don't have to buy all of the fruits. You store them in a dictionary. If you don't have the product, it disappears from your dictionary. A client can come in your shop and buy fruits if there are enough.
You'll create four functions:

1. `buy_fruit_from_wholesaler`
2. `sell_fruit_to_client`
3. `print_wholesaler`
4. `print_storage`

Function `print_storage`

```
1  # -*- coding: utf-8 -*-
2
3  def print_storage(storage_dict):
4      """
5      Function to display the storage of our fruits shop
6      If the storage_dict is empty, print: 'Sorry, no
           article available'. Else print the details of the
           shop.
7
8      Return nothing
9      """
10
11     if storage_dict == {}:
12         print('Sorry, no article available.\n')
13     else:
14         print('Fruits available :\n--------------------')
15         for fruit, number in storage_dict.items():
16             print(' -',fruit, number)
17         print('\n')
```

Function print_wholesaler

```
def print_wholesaler(wholesaler_list):
    """
    Function to display the available fruits in the
       wholesaler

    Return nothing
    """

    print('Welcome to the wholesaler! Here\'s the fruits
       available in infinite quantity:')

    for i, element in enumerate(wholesaler_list):
        print(i,':',element)
    print('\n')
```

Function sell_fruit_to_client

```
def sell_fruit_to_client(storage_dict, fruit, quantity):
    """
    Function to sell to a client a fruit in a certain
       amount quantity.
    """

    if fruit not in storage_dict:
        print('Sorry, we don\'t have this fruit.\n')
    else:
        storage_quantity = storage_dict.get(fruit)
        if quantity > storage_quantity:
            print('Sorry, I cannot give you {} {}. There
               are only {} left.\n'.format(quantity,
               fruit, storage_quantity))
        elif quantity == storage_quantity:
            print('Here your {} {}\n'.format(quantity,
               fruit))
            storage_dict.pop(fruit)
        else:
            print('Here your {} {}\n'.format(quantity,
               fruit))
            storage_dict[fruit] -= quantity
```

Function buy_fruit_from_wholesaler

```
def buy_fruit_from_wholesaler(fruits_list, storage_dict,
    fruit, quantity):
    """
    Function to buy a certain fruit in the quantity you
        want and add it to you storage_dict.
    """

    if fruit not in fruits_list:
        print('Sorry, the fruit you asked is not
            available.\n')
    else:
        if quantity < 1:
            print('Please, choose a positive integer.\n')
        elif fruit in storage_dict:
            storage_dict[fruit] += quantity
        else:
            storage_dict[fruit] = quantity
```

Main script to test the functions

```
### Here we test the functions ...
if __name__=='__main__':

    # our shop is here :
    fruits_shop = {}

    # The wholesaler list is here
    wholesaler_list = ['orange', 'apple', 'grape', '
       watermelon', 'peach', 'banana', 'apricot', 'cherry
       ']

    print_storage(fruits_shop) # Nothing for the moment

    buy_fruit_from_wholesaler(wholesaler_list,
       fruits_shop, 'banana', 25)
    buy_fruit_from_wholesaler(wholesaler_list,
       fruits_shop, 'apple', 17)
    buy_fruit_from_wholesaler(wholesaler_list,
       fruits_shop, 'watermelon', 91)
    buy_fruit_from_wholesaler(wholesaler_list,
       fruits_shop, 'avocado', 10) # fruit isn't in the
       wholesaler list

    sell_fruit_to_client(fruits_shop, 'banana', 5)
    sell_fruit_to_client(fruits_shop, 'apple', 17)
    sell_fruit_to_client(fruits_shop, 'cherry', 40) #
       fruit isn't in the storage
    sell_fruit_to_client(fruits_shop, 'watermelon', 92) #
        too much, he cannot buy it

    print_storage(fruits_shop) # you normally get 20
       bananas and 91 watermelons.
```

```
Welcome to my supermarket! Here's what we have today:
Sorry, no article available.

Welcome to the wholesaler! Here's the fruits available in
   infinite quantity:
0 : orange
1 : apple
2 : grape
3 : watermelon
4 : peach
```

```
5 : banana
6 : apricot
7 : cherry


Please, choose a positive integer.

Sorry, the fruit you asked is not available.

Welcome to my supermarket! Here's what we have today:
Fruits available :
--------------------
 - banana 25
 - apple 17
 - watermelon 91


Here your 5 banana

Here your 17 apple

Sorry, we don't have this fruit.

Sorry, I cannot give you 92 watermelon. There are only 91
   left.

Welcome to my supermarket! Here's what we have today:
Fruits available :
--------------------
 - banana 20
 - watermelon 91
```

6.2.2 Karaoke Bar

data.py

```
max_songs = 20

# dictionary of songs
songs = {
    1: 'Guys - Chi minii',
    2: 'Aqua - Barbie Girl',
    3: 'Rihanna - Umbrella',
    4: 'Whitney Houston - I will always love you',
    5: 'Michael Jackson - Thriller',
    6: 'Ginjin and Mrs M - Boroo',
    7: 'Clean Bandit - Symphony',
    8: 'Stromae - Papaoutai',
    9: 'Avicii - Levels',
    10: 'Earth Wind and Fire - September',
    11: 'ABBA - Dancing Queen',
    12: 'Haddaway - What is Love',
    13: 'Men at Work - Down Under',
    14: 'Patrick Sebastien - Les Sardines',
    15: 'Beyonce - Love on Top',
    16: 'Clean Bandit - Solo',
    17: 'Sexy Sushi - Cheval',
    18: 'Big Boi - All night',
    19: 'Little Big - Everyday Im drinking',
    20: 'Rick Astley - Never gonna give you up'
}

# dictionary of alcohol
alcohol_unit = {
    'Niislel': 4500,
    'Craft': 5500,
    'Airag': 3500,
    'Eden': 2000
}

alcohol_bottle = {
    'Eden': 40000,
    'Red wine': 60000,
    'White wine': 55000
}
```

functions.py

```
# -*- coding: utf-8 -*-
"""
Created on Fri Feb  5 10:45:22 2021

@author: jeremow

Useful functions for the karaoke bar algorithm
"""

from data import *
from random import choice


def create_users(max_users=5):
    """
    Ask the leader how many there are and the names of
       each.

    Returns
    -------
    users: dict.

    """

    users = {}

    print('Welcome to UB KARAOKE\n-------------')

    nb_users = int(input('How many people will sing? (Min
       : 1, Max: {}) :\n'.format(max_users)))

    while nb_users < 1 or nb_users > max_users:
        nb_users = int(input('Sorry, not possible. How
           many people will sing? (Min: 1, Max: {})'.
           format(max_users)))

    print('\nHere we go for their names, give one by one.
       ')

    for i in range(1, nb_users + 1):
        name = input('User {}: '.format(i))
        users[name] = 0

    print('\nThank you and have a great evening.')
```

```
40
41      return users
42
43
44  def add_songs(waiting_list, count_songs):
45      """
46      Adding songs to the waiting list of the karaoke
47
48      Parameters
49      ----------
50      waiting_list : TYPE
51          DESCRIPTION.
52      count_songs : TYPE
53          DESCRIPTION.
54
55      Returns
56      -------
57      None.
58
59      """
60
61      print('The list of songs :\n-----------')
62
63      for key, value in songs.items():
64          print(key, ':', value)
65
66      nb_songs = len(waiting_list) + count_songs
67      remaining_songs = max_songs - nb_songs
68
69      nb_add_songs = int(input(
70          'There are {} remaining songs before leaving.\
               nHow many songs to you want to add? '.format(
               remaining_songs)))
71
72      while nb_add_songs < 0 or nb_add_songs >
           remaining_songs:
73          nb_add_songs = int(input(
74              'Sorry, value is not possible.\nThere are {}
                   remaining songs before leaving.\n'
75              'How many songs to you want to add? '.format(
                   remaining_songs)))
76
77      i = 0
78      while i < nb_add_songs:
79          try:
80              i_song = int(input('Write the number of the
```

```
                song (0 for random): '))
81              if i_song == 0:
82                  waiting_list.append(choice(songs))
83              else:
84                  waiting_list.append(songs[i_song])
85              i += 1
86          except KeyError:
87              print('Sorry, the number of song is incorrect
                .')
88          except ValueError:
89              print('Sorry, this is not a number.')
90
91      print('\nWaiting list is now:\n----------')
92
93      for song in waiting_list:
94          print(song)
95
96
97  def order_drinks(users):
98      """
99      Order drinks to the barman/maid
100
101     Parameters
102     ----------
103     users : DICT
104         DESCRIPTION.
105
106     Returns
107     -------
108     None.
109
110     """
111
112     drink_answer = input('Do you want to order a drink? [
            y]/n ')
113
114     if drink_answer.lower() == 'n':
115         print('See you in three songs!\n')
116
117     else:
118
119         nb_users = len(users)
120         bottle_answer = input('Do you want a bottle ? [n
                ]/y ')
121
122         if bottle_answer.lower() == 'y':
```


```
123             print('Here\'s the list of bottles we have :\
                    n----------')
124
125             for key, value in alcohol_bottle.items():
126                 print('{}: {} Tugrug'.format(key, value))
127
128             order = True
129
130             while order:
131                 try:
132                     bottle_name = input('Write the name
                            of the bottle (case sensitive): ')
133                     price_bottle = alcohol_bottle[
                            bottle_name]
134
135                     split_price = int(round(price_bottle
                            / nb_users))
136
137                     for key in users.keys():
138                         users[key] += split_price
139
140                     order = False
141
142                 except KeyError:
143                     print('Sorry, this bottle doesn\'t
                            exist.\n')
144
145         drink_answer = input('Do you want single drinks ?
                [y]/n ')
146
147         if drink_answer != 'n':
148             print('Here\'s the list of drinks we have :\n
                    ----------')
149
150             for key, value in alcohol_unit.items():
151                 print('{}: {} Tugrug'.format(key, value))
152
153             for key in users.keys():
154                 try:
155                     drink_name = input(
156                         'What do you want to drink, {} ?
                                (enter for nothing, case
                                sensitive):\n'.format(key))
157                     price_drink = alcohol_unit[drink_name
                            ]
158                     users[key] += price_drink
```

```
159
160                 except KeyError:
161                     pass
162
163
164 def get_bill(users):
165     """
166     Get the bill for everyone in the room of the karaoke
        bar.
167
168     Parameters
169     ----------
170     users : DICT
171         DESCRIPTION.
172
173     Returns
174     -------
175     total_bill : INT.
176
177     """
178
179     total_bill = 0
180
181     print('\nThe bill of you evening\n-----------')
182
183     for key, value in users.items():
184         print('{}: {} Tugrug'.format(key, value))
185
186         total_bill += value
187
188     return total_bill
189
190
191 if __name__ == '__main__':
192     waiting_list = []
193
194     add_songs(waiting_list, 1)
195
196     users = create_users()
197     order_drinks(users)
```


karaokebar.py

```
from functions import *
from data import *

# Variables to exit the while-loop
count_songs = 0
stay_answer = True

# Initialize the list for songs
waiting_list = []

# Initialize the dict for users, we will save the bill
    next to the name
users = create_users()

while count_songs < max_songs and stay_answer:

    add_songs(waiting_list, count_songs)

    if count_songs % 3 == 0:
        order_drinks(users)

    try:
        print('\nNext song is {}, get ready!'.format(
            waiting_list[0]))
        waiting_list.pop(0)
        count_songs += 1
    except IndexError:
        print('\nThere is no song in the waiting list !\n
            ')

    input('Push enter to finish the song...')

    stay_answer = input('Do you want to continue to sing?
        [y]/n ')

    if stay_answer.lower() == 'n':
        stay_answer = False
    else:
        stay_answer = True

total_price = get_bill(users)

print('Thank you for your visit and see you soon!')
```

6.3. ObsPy

6.3.1 Timestamps

Exercise 1

Calculate the number of days passed between today and the Tohoku main shock (`2011-03-11T05:46:23.2`).

```
1 from obspy import UTCDateTime
2
3 nb_days = (UTCDateTime() - UTCDateTime("2011-03-11T05
    :46:23.200000Z"))/86400
4 print(round(nb_days))
```

```
3826
```

Exercise 2

Make a list of 10 `UTCDateTime` objects, starting today at 10:00 with a spacing of 90 minutes.

```
1 times = []
2 current_time = UTCDateTime("2021-06-03T10:00:00.000000Z")
3 for i in range(0, 10):
4     times.append(current_time + 90*60*i)
5
6 print(times)
```

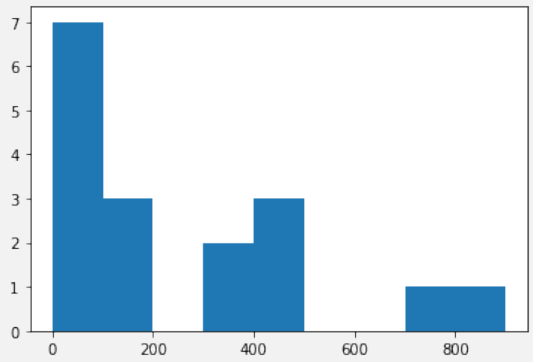
```
[ UTCDateTime(2021, 6, 3, 10, 0),
  UTCDateTime(2021, 6, 3, 11, 30),
  UTCDateTime(2021, 6, 3, 13, 0),
  ...,
  UTCDateTime(2021, 6, 3, 20, 30),
  UTCDateTime(2021, 6, 3, 22, 0),
  UTCDateTime(2021, 6, 3, 23, 30) ]
```


Exercise 3

Below is a list of strings with origin times of magnitude 8+ earthquakes between 2000 and 2013 (fetched from IRIS). Assemble a list of inter-event times in days. Use matplotlib to display a histogram.

```
import matplotlib.pyplot as plt

times = ["2001-06-23T20:33:09", "2003-09-25T19:50:07",
         "2004-12-23T14:59:00", "2004-12-26T00:58:52",
         "2005-03-28T16:09:35", "2006-06-01T18:57:02",
         "2006-06-05T00:50:31", "2006-11-15T11:14:14",
         "2007-01-13T04:23:23", "2007-04-01T20:39:56",
         "2007-08-15T23:40:58", "2007-09-12T11:10:26",
         "2009-09-29T17:48:11", "2010-02-27T06:34:13",
         "2011-03-11T05:46:23", "2012-04-11T08:38:36",
         "2012-04-11T10:43:10", "2013-05-24T05:44:48"]

interevent = []
for i in range(0, len(times)-1):
    interevent.append((UTCDateTime(times[i+1])-
       UTCDateTime(times[i]))/86400)
print(interevent)

plt.hist(interevent, bins=range(0, 1000, 100))
```

```
[ 823.9701157407408, 454.7978356481481,
  2.416574074074074, ..., 397.1195949074074,
  0.08650462962962963, 407.7928009259259 ]
```



6.3.2 Traces

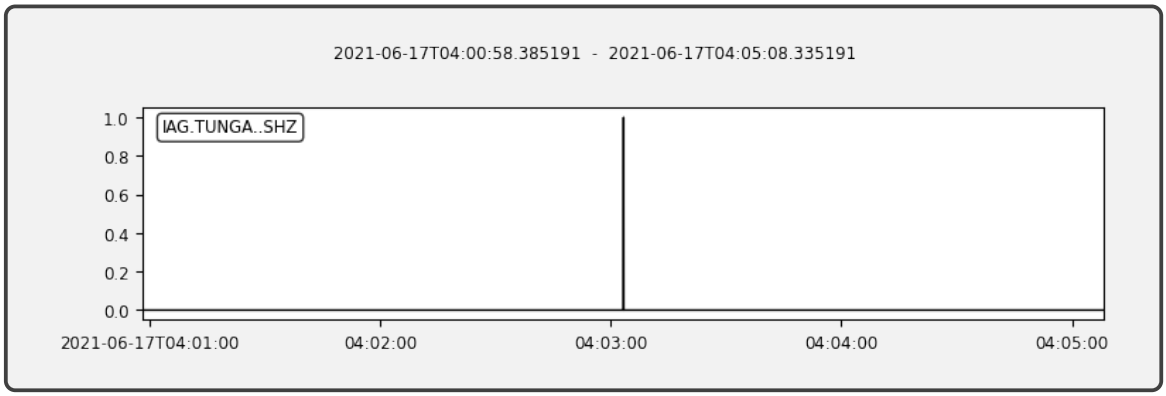
The four parts constitutes a single exercise.

Part 1

1. Make an `numpy.ndarray` with zeros and (use `numpy.zeros()`) and put an ideal pulse somewhere in it
2. Initialize a Trace object with your data array
3. Fill in some station information (`network`, `station`, ...)
4. Print trace summary and plot the trace
5. Change the sampling rate to 20 Hz
6. Change the `starttime` of the `Trace` to the start time of this session
7. Print the `Trace` summary and plot the `Trace` again

```
1  import numpy as np
2  from obspy import Trace, UTCDateTime
3
4  x = np.zeros(5000)
5  x[2500] = 1.0
6
7  tr = Trace(data=x)
8  tr.stats.sampling_rate = 20.0
9
10 tr.stats.network = 'IAG'
11 tr.stats.station = 'TUNGA'
12 tr.stats.channel = 'SHZ'
13
14 starttime = UTCDateTime()
15
16 tr.stats.starttime = starttime
17
18 print(tr)
19 tr.plot()
```

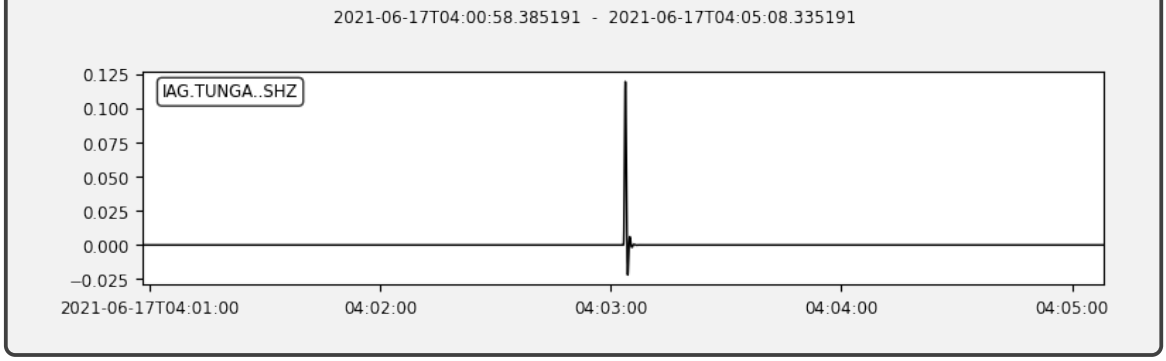
```
IAG.TUNGA..SHZ | 2021-06-17T04:00:58.385191Z - 2021-06-17
   T04:05:08.335191Z | 20.0 Hz, 5000 samples
```



Part 2

1. `Use tr.filter(...)` and apply a lowpass filter with a corner frequency of 1 Hz.
2. Display the plot, there are a few seconds of zeros that we can cut off.

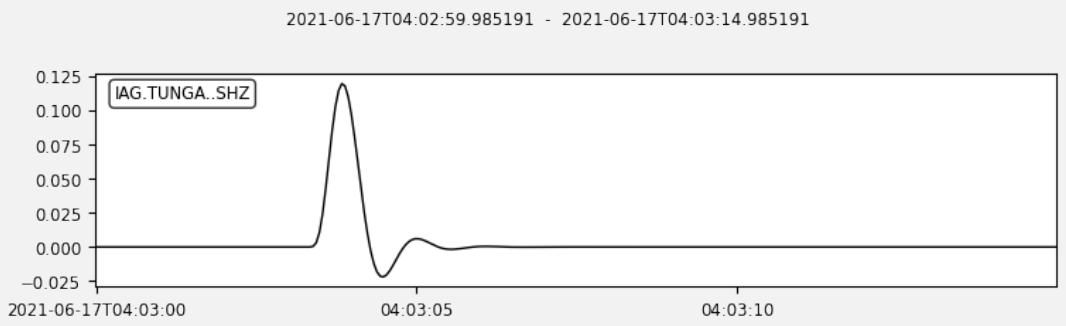
```
1 tr.filter(type='lowpass', freq=1.0)
2 tr.plot()
```



Part 3

1. Use `tr.trim(...)` to remove some of the zeros at start and at the end
2. Show the plot again

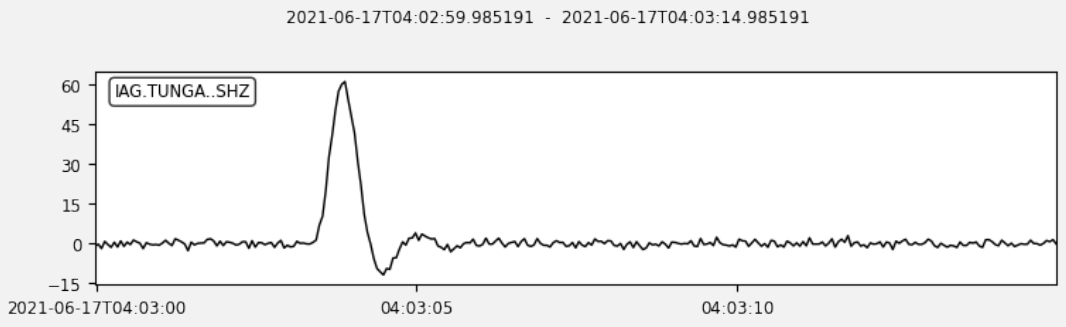
```
t1 = UTCDateTime("2021-06-17T04:03:00")
t2 = UTCDateTime("2021-06-17T04:03:15")
tr.trim(starttime=t1, endtime=t2)
tr.plot()
```



Part 4

1. Scale up the amplitudes of the trace by a factor of 500
2. Add standard normal Gaussian noise to the `Trace` (use `np.random.randn()`)
3. Display the plot again

```
tr.data *= 500
tr.data = tr.data + np.random.randn(len(tr.data))
tr.plot()
```



6.3.3 Stream

The four parts constitutes a single exercise. The data used are available on GitHub in the `data` folder: https://github.com/jeremow/Python_lessons/tree/main/data.

Part 1

1. Read all Tohoku example earthquake data into a `Stream` object (`./data/waveform_*`)

2. Print the stream summary

```
from obspy import read

st = read("./data/waveform_*")
print(st)
```

```
8 Trace(s) in Stream:
GR.BFO..BHE   | 2011-03-11T05:46:23.021088Z - 2011-03-11
   T06:36:22.971088Z | 20.0 Hz, 60000 samples
GR.BFO..BHN   | 2011-03-11T05:46:23.021088Z - 2011-03-11
   T06:36:22.971088Z | 20.0 Hz, 60000 samples
GR.BFO..BHZ   | 2011-03-11T05:46:23.021088Z - 2011-03-11
   T06:36:22.971088Z | 20.0 Hz, 60000 samples
II.PFO.00.BHZ | 2011-03-11T05:46:23.019500Z - 2011-03-11
   T06:36:22.969500Z | 20.0 Hz, 60000 samples
II.PFO.10.BHZ | 2011-03-11T05:46:23.019500Z - 2011-03-11
   T06:36:22.994500Z | 40.0 Hz, 120000 samples
SY.PFO.S3.MXE | 2011-03-11T05:47:31.587750Z - 2011-03-11
   T06:36:22.974250Z | 6.2 Hz, 18152 samples
SY.PFO.S3.MXN | 2011-03-11T05:47:31.587750Z - 2011-03-11
   T06:36:22.974250Z | 6.2 Hz, 18152 samples
SY.PFO.S3.MXZ | 2011-03-11T05:47:31.587750Z - 2011-03-11
   T06:36:22.974250Z | 6.2 Hz, 18152 samples
```

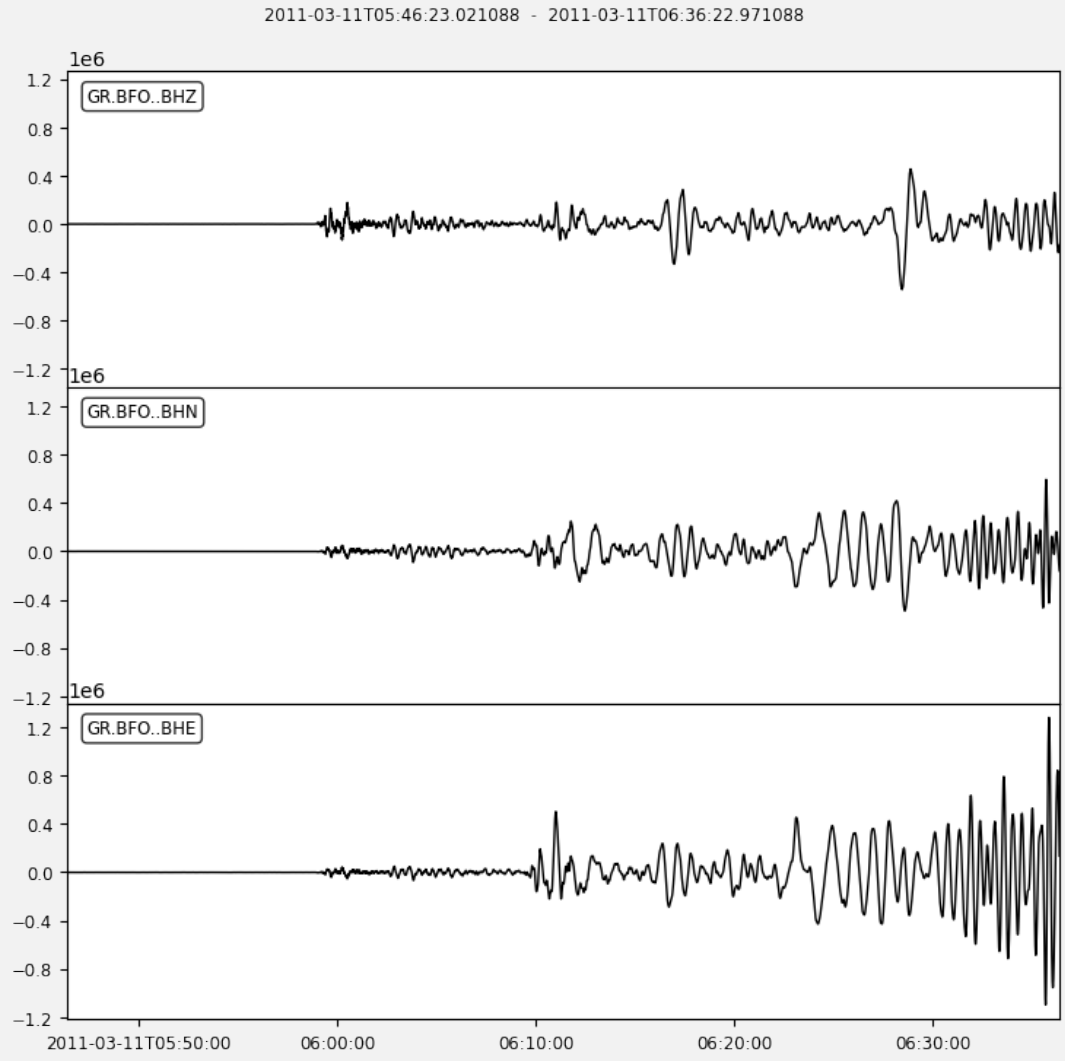
Part 2

1. Use `st.select()` to only keep traces of station BFO in the `Stream`

2. Display the plot

```
st = st.select(station='BFO')
print(st)
st.plot()
```

```
3 Trace(s) in Stream:
GR.BFO..BHE | 2011-03-11T05:46:23.021088Z - 2011-03-11T06
   :36:22.971088Z | 20.0 Hz, 60000 samples
GR.BFO..BHN | 2011-03-11T05:46:23.021088Z - 2011-03-11T06
   :36:22.971088Z | 20.0 Hz, 60000 samples
GR.BFO..BHZ | 2011-03-11T05:46:23.021088Z - 2011-03-11T06
   :36:22.971088Z | 20.0 Hz, 60000 samples
```

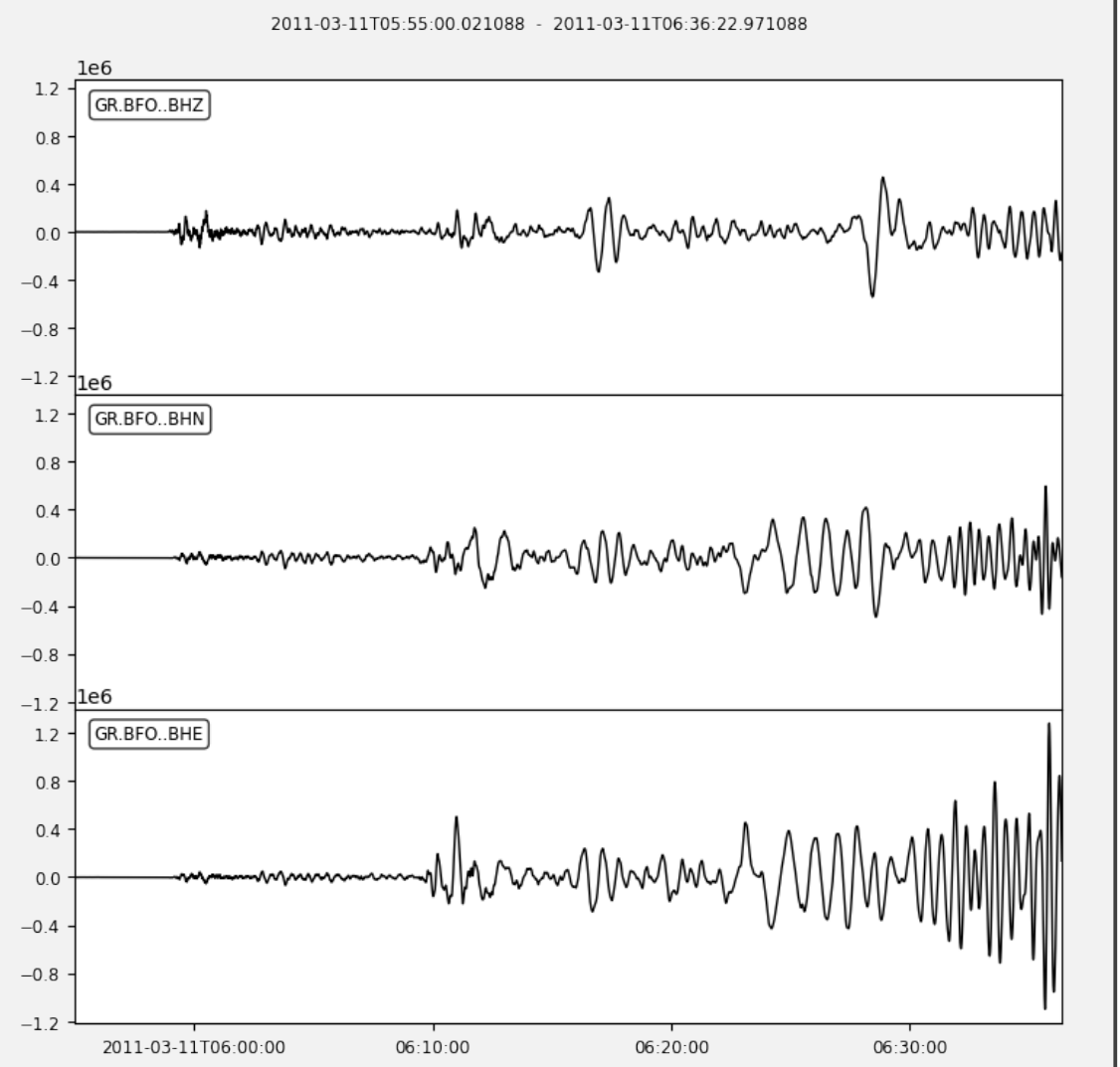


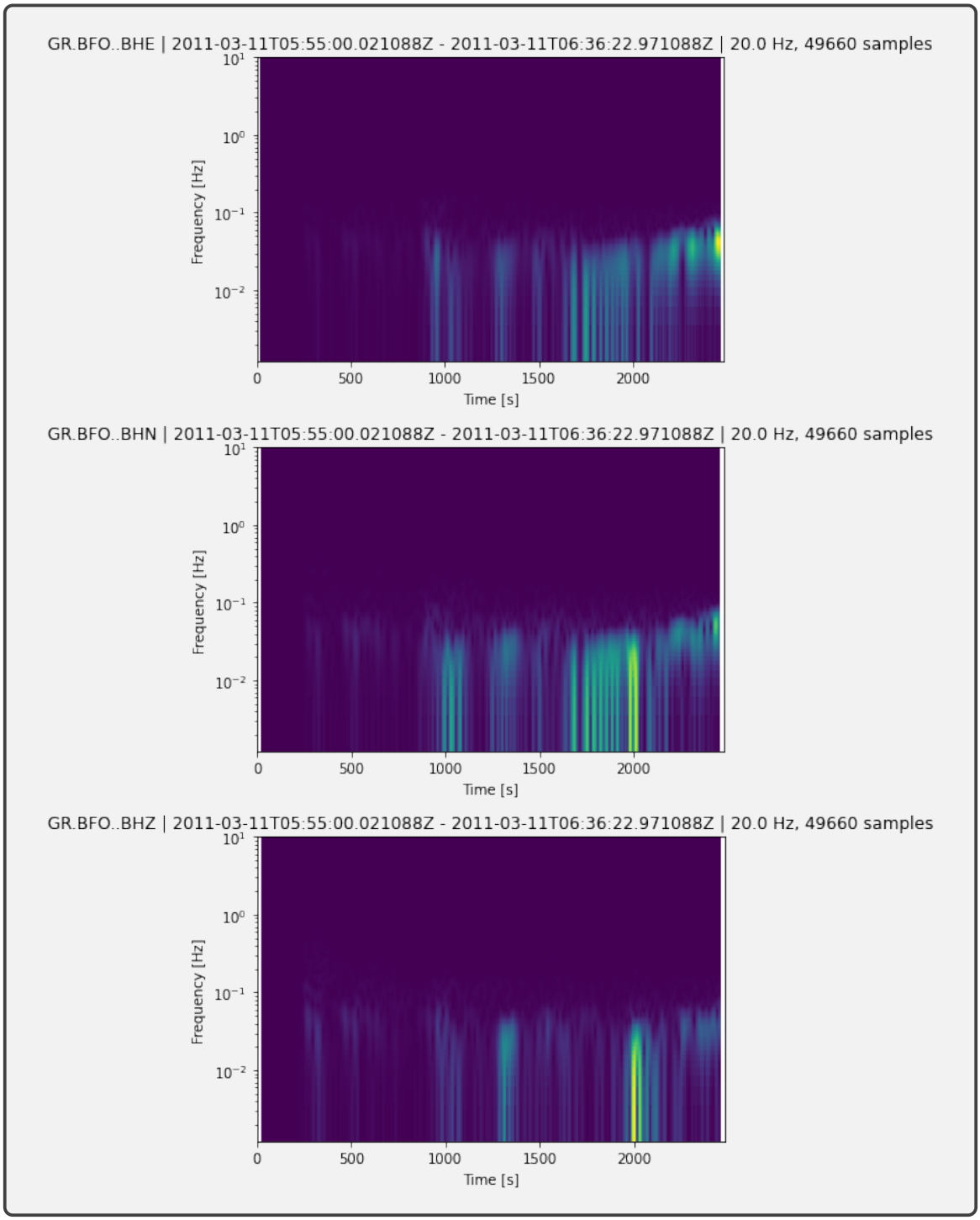
Part 3

1. Trim the data to a 10 minute time window around the first arrival (just roughly looking at the previous plot)

2. Display the plot

```
1 from obspy import UTCDateTime
2
3 st.trim(starttime=UTCDateTime("2011-03-11T05:55:00"))
4 st.plot()
5 st.spectrogram(log=True, wlen=50)
```



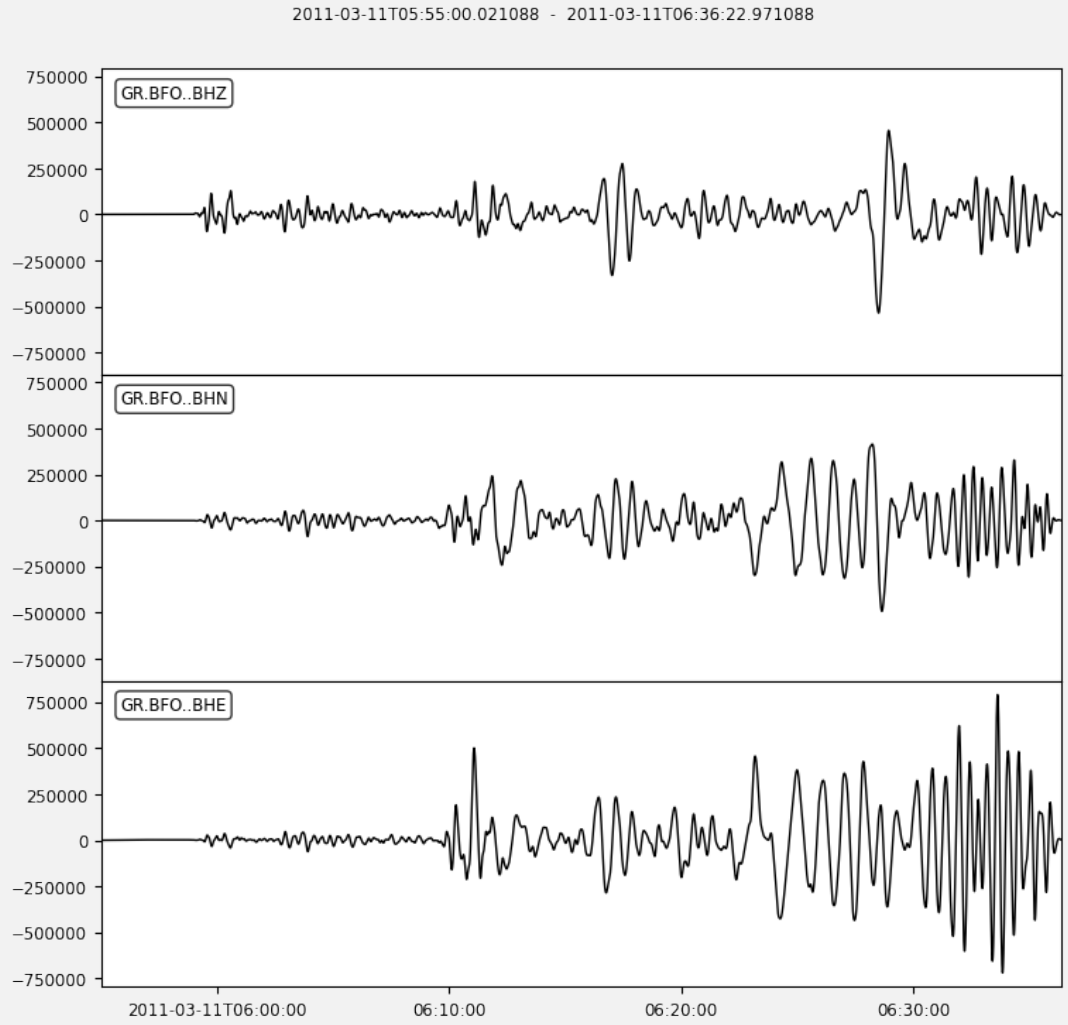

GR.BFO..BHE | 2011-03-11T05:55:00.021088Z - 2011-03-11T06:36:22.971088Z | 20.0 Hz, 49660 samples
Frequency [Hz]
Time [s]
GR.BFO..BHN | 2011-03-11T05:55:00.021088Z - 2011-03-11T06:36:22.971088Z | 20.0 Hz, 49660 samples
Frequency [Hz]
Time [s]
GR.BFO..BHZ | 2011-03-11T05:55:00.021088Z - 2011-03-11T06:36:22.971088Z | 20.0 Hz, 49660 samples
Frequency [Hz]
Time [s]

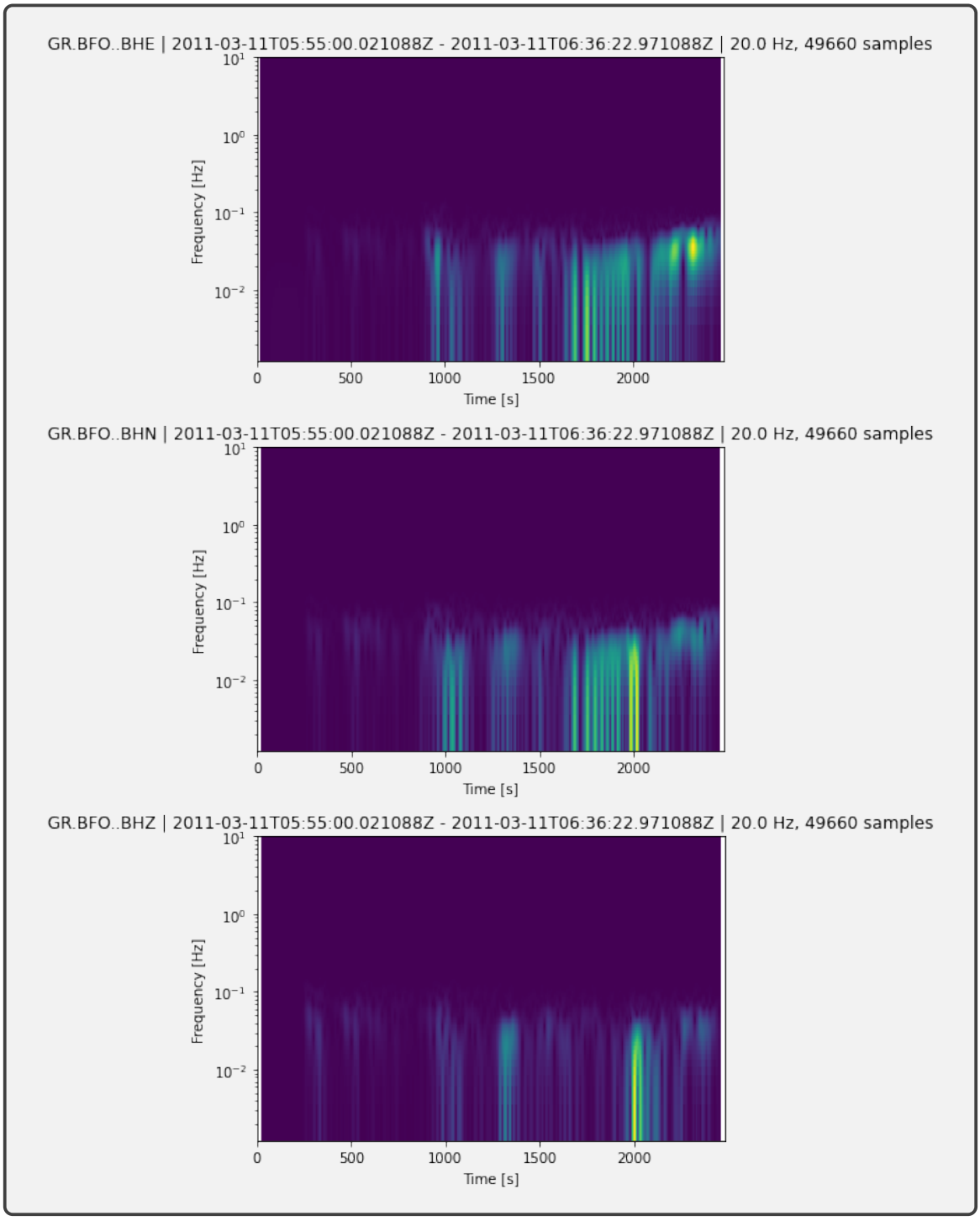
Part 4

1. Remove the linear trend from the data, apply a tapering and a lowpass at 0.1 Hertz

2. Display the plot

```
st.detrend('linear')
st.taper(type="hann", max_percentage=0.05)
st.filter("lowpass", freq=0.1)
st.plot()
st.spectrogram(log=True, wlen=50)
```




GR.BFO..BHE | 2011-03-11T05:55:00.021088Z - 2011-03-11T06:36:22.971088Z | 20.0 Hz, 49660 samples
Frequency [Hz]
Time [s]
GR.BFO..BHN | 2011-03-11T05:55:00.021088Z - 2011-03-11T06:36:22.971088Z | 20.0 Hz, 49660 samples
Frequency [Hz]
Time [s]
GR.BFO..BHZ | 2011-03-11T05:55:00.021088Z - 2011-03-11T06:36:22.971088Z | 20.0 Hz, 49660 samples
Frequency [Hz]
Time [s]

6.3.4 FDSN Client

Use the FDSN client to assemble a waveform dataset for an event. The four parts constitutes a single exercise.

Part 1

Search for a large earthquake on IRIS (by depth or in a region of your choice, use option `limit=5` to keep network traffic down).

```
from obspy.clients.fdsn import Client

client = Client("IRIS")
catalog = client.get_events(minmagnitude=7, mindepth=400,
    limit=5)
print(catalog)

event = catalog[0]
print(event)
```

```
5 Event(s) in Catalog:
2018-09-06T15:49:14.420000Z | -18.495, +179.356 | 7.9 mww
2018-08-24T09:04:06.780000Z | -11.042,  -70.817 | 7.1 Mww
2018-08-19T00:19:40.670000Z | -18.113, -178.154 | 8.2 mww
2017-01-10T06:13:47.250000Z |  +4.463, +122.575 | 7.3 Mww
2015-11-24T22:50:54.370000Z | -10.060,  -71.018 | 7.6 mww
Event:  2018-09-06T15:49:14.420000Z | -18.495, +179.356 |
   7.9 mww

               resource_id: ResourceIdentifier(id="smi:
                  service.iris.edu/fdsnws/event/1/query?
                  eventid=10944928")
                event_type: 'earthquake'
        preferred_origin_id: ResourceIdentifier(id="smi:
           service.iris.edu/fdsnws/event/1/query?originid
           =34433936")
     preferred_magnitude_id: ResourceIdentifier(id="smi:
        service.iris.edu/fdsnws/event/1/query?magnitudeid
        =189562093")
                    ---------
         event_descriptions: 1 Elements
                    origins: 1 Elements
                 magnitudes: 1 Elements
```

Part 2

Search for stations to look at waveforms for the event. The stations should:

1. Be available at the time of the event
2. Have a vertical 1 Hz stream ("LHZ" for example)
3. Be in a narrow angular distance around the event (e.g. 90-91 degrees)
4. Adjust your search so that only a small number of stations (3-6) match your search criteria

```
print(event.origins)

origin = event.origins[0]
t = origin.time
lon = origin.longitude
lat = origin.latitude

inventory = client.get_stations(longitude=lon,
                    latitude=lat,
                    minradius=90.5, maxradius=90.8,
                    starttime=t, endtime =t+100,
                    channel="LHZ", matchtimeseries=True)
print(inventory)
```

```
[Origin
       resource_id: ResourceIdentifier(id="smi:service.
          iris.edu/fdsnws/event/1/query?originid
          =34433936")
               time: UTCDateTime(2018, 9, 6, 15, 49, 14,
                  420000)
          longitude: 179.3555
           latitude: -18.4952
              depth: 617870.0
     creation_info: CreationInfo(author='at,pt,us')]

Inventory created at 2021-06-24T07:53:24.000000Z
    Created by: IRIS WEB SERVICE: fdsnws-station | version
       : 1.1.47
            http://service.iris.edu/fdsnws/station/1/query
               ?starttime=2018-09-06...
    Sending institution: IRIS-DMC (IRIS-DMC)
    Contains:
        Networks (5):
            7C, C1, N4, TA, US
```

```
        Stations (6):
            7C.MM27 (Macmillan Pass, YT, Canada)
            C1.LR04 (Corral)
            N4.MSTX (Muleshoe, TX, USA)
            TA.D24K (Happy Valley, AK, USA)
            TA.T25A (Trinidad, CO, USA)
            US.RLMT (Red Lodge, Montana, USA)
        Channels (0):
```

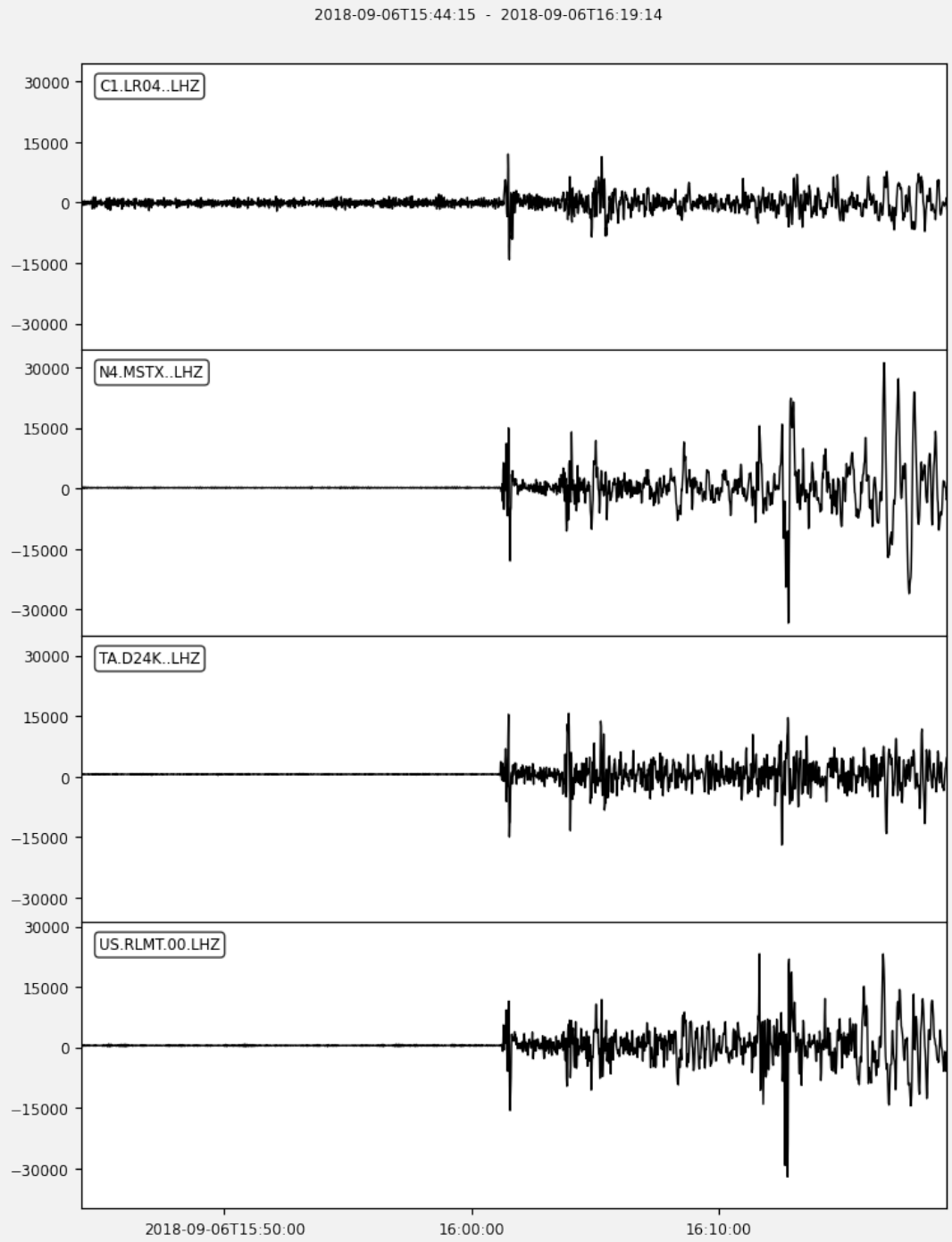
Part 3

1. For each of these stations download data of the event, like a couple of minutes before to half an hour after the event
2. Put all data together in one `Stream` (put the `get_waveforms()` call in a `try/except` block to silently skip stations that actually have no data available)
3. Print stream info, plot the raw data

```
1  from obspy import Stream
2  st = Stream()
3
4  for network in inventory:
5      for station in network:
6          try:
7              st += client.get_waveforms(network.code,
8                  station.code, "*", "LHZ",
9                  t - 5 * 60, t + 30 * 60,
10                 attach_response=True)
11         except:
12             print('No data')
13
14 print(st)
15 st.plot()
```

```
No data
No data
4 Trace(s) in Stream:
C1.LR04..LHZ   | 2018-09-06T15:44:15.000000Z - 2018-09-06
   T16:19:14.000000Z | 1.0 Hz, 2100 samples
N4.MSTX..LHZ   | 2018-09-06T15:44:15.000000Z - 2018-09-06
   T16:19:14.000000Z | 1.0 Hz, 2100 samples
TA.D24K..LHZ   | 2018-09-06T15:44:15.000000Z - 2018-09-06
```

```
   T16:19:14.000000Z | 1.0 Hz, 2100 samples
US.RLMT.00.LHZ | 2018-09-06T15:44:15.000000Z - 2018-09-06
   T16:19:14.000000Z | 1.0 Hz, 2100 samples
```



Part 4

Correct the instrument response for all stations with a `water_level=20` and plot the corrected data.

```
st.remove_response(water_level=20)
st.plot()
```