Generators

Generators are data structures similar to lists with the difference that in memory it does not hold all the values of the list, but instead, the computer code needed to well.. generate them.

Although we have not emphasized it, the `range` , `enumerate` , `items` , `map` that we have presented before, in essense, return a generator.

To better understand generators let's use a real life example. Suppose someone tells you: memorize the following 100 phone numbers in that given order. After that, say the first number, then the second, etc ..

This would require a lot of effort (and memory) from you.

Suppose now that someone tells you: memorize all even numbers from 2 to 1000. After that tell me the first then the second, etc ..

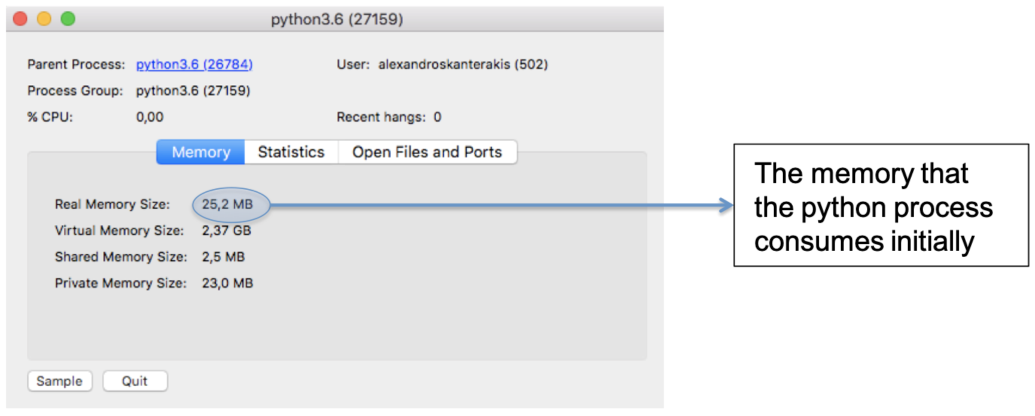
No special effort is required for this. You just have to remember the logic in order to get the next number in this sequence. This is exactly what generators do. They are useful in cases where the next item in a list can be computed and not recalled from memory.

So with the following command

```
range(1,100_000_000)
```

There is no reason to store 100,000,000 values in computer memory. Let's see it in practice. If we run:

```
a = [x for x in range(1,100_000_000)]
```

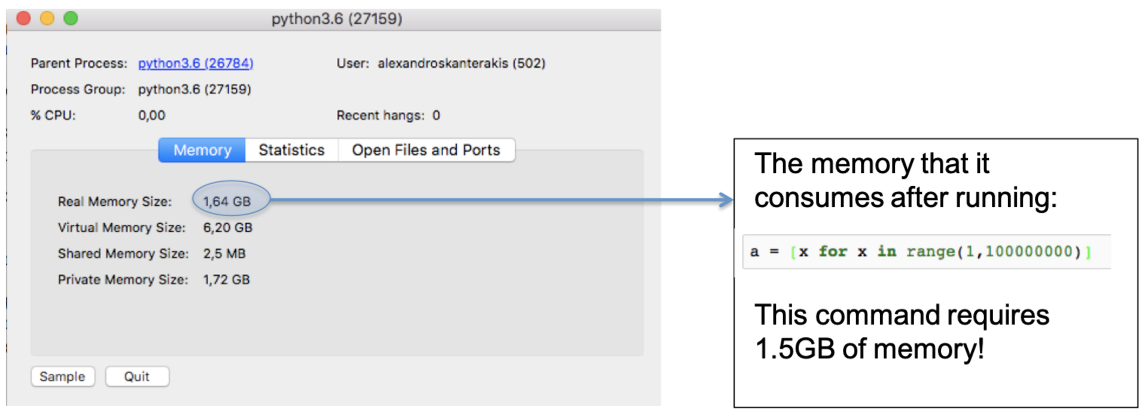


python3.6 (27159)

Parent Process: python3.6 (26784)   User: alexandroskanterakis (502)
Process Group: python3.6 (27159)
% CPU: 0,00   Recent hangs: 0

Real Memory Size: 25,2 MB
Virtual Memory Size: 2,37 GB
Shared Memory Size: 2,5 MB
Private Memory Size: 23,0 MB

The memory that the python process consumes initially



python3.6 (27159)

Parent Process: python3.6 (26784)   User: alexandroskanterakis (502)
Process Group: python3.6 (27159)
% CPU: 0,00   Recent hangs: 0

Real Memory Size: 1,64 GB
Virtual Memory Size: 6,20 GB
Shared Memory Size: 2,5 MB
Private Memory Size: 1,72 GB

The memory that it consumes after running:

```
a = [x for x in range(1,100000000)]
```

This command requires 1.5GB of memory!

We see that python needs 1.5GB of memory. If we need to get arbitrary values from list `a` then we can not do otherwise. Otherwise if we just need to get one value after another then we can save this memory by writing:

```
for x in range(1, 1000000):
    ...
```

Note: Is this important? This is very important. Computer programming is all about solving problems with given and finite resources. A computer programming is **not** a theoretical construct in a piece of paper. This is an algorithm. A good engineer should provide solutions that minimize: resources / time / complexity.

We can create our own generator in two ways: The first is through generator comprehension and the second is through functions that instead of `return` do `yield` .

The syntax if generator comprehension is just like list comprehensions (and dictionary, set comprehension) with the only difference that we just use parentheses:

In [1]:
```
my_generator = (x for x in range(1,10))
type(my_generator)
```

Out[1]:
```
generator
```

Let's get the first value of the generator:

In [2]:
```
next(my_generator)
```

Out[2]:
```
1
```

Let's take the next one:

In [3]:
```
next(my_generator)
```

Out[3]:
```
2
```

... etc .. We can get all the rest values of the generator:

In [4]:
```
for x in my_generator:
    print (x)
```

```
3
4
5
6
7
8
9
```

We see that the generator "remembers" on which what value we were at, and calculates the next one. When the generator runs out of values then it raises an exeption:

In [5]:
```
next(my_generator)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-5-beb7403f481d> in <module>
----> 1 next(my_generator)
```

```
StopIteration:
```

The second way to make a generator is to declare a function with `def` and instead of `return` to do `yield` :

```
In [6]: def f():
            yield "first"
            yield "second"
            yield "third"
```

Attention `f` is a function that makes a generator! `f` is not a generator!

```
In [7]: type(f)

Out[7]: function
```

```
In [8]: my_generator = f()
        type(my_generator)

Out[8]: generator
```

`my_generator` is a generator. Let's "run" it:

```
In [9]: next(my_generator)

Out[9]: 'first'
```

```
In [10]: next(my_generator)

Out[10]: 'second'
```

```
In [11]: next(my_generator)

Out[11]: 'third'
```

Or else:

```
In [12]: for x in f():
             print (x)

first
second
third
```

Another example: A generator that generates prime numbers:

```
In [13]: def prime_gen():
             yield 1
             n=2
             while True:
                 for d in range(2, n-1):
                     if n%d==0:
                         break
                 else:
                     yield n
                 n+=1
```

```
In [14]: g = prime_gen()
```

```
In [15]: next(g)
```

```
Out[15]: 1
```

```
In [16]: next(g)
```

```
Out[16]: 2
```

```
In [17]: next(g)
```

```
Out[17]: 3
```

```
In [18]: next(g)
```

```
Out[18]: 5
```

```
In [19]: g = prime_gen()
         for i in range(10):
             print (next(g))
```

```
1
2
3
5
7
11
13
17
19
23
```

When a generator "terminates", it throws an exception (we will see later more) which we can catch:

```
In [21]: gen = (x*2 for x in range(10))

         while True:
             try:
                 n = next(gen)
             except StopIteration:
                 break
             print (n)
```

```
0
2
4
6
8
10
12
14
16
18
```

import

With `import` we can well.. import code in our environment from another file. Suppose we have the `a.py` file which has the following code:

```
# File: a.py

def f():
    print ("hello")

def g():
    print ("world")
```

We can create the file `a.py` by "running" the following cell:

In [2]:
```
%%writefile a.py
def f():
    print ("hello")

def g():
    print ("world")

name="mitsos"
```

```
Writing a.py
```

Now we can `import` these functions:

In [3]:
```
from a import f # Directly importing f
f()
```

```
hello
```

In [4]:
```
from a import f,g # Directly importing f and g
f()
g()
```

```
hello
world
```

In [5]:
```
from a import * # Importing everything that exist in a.py
f()
g()
```

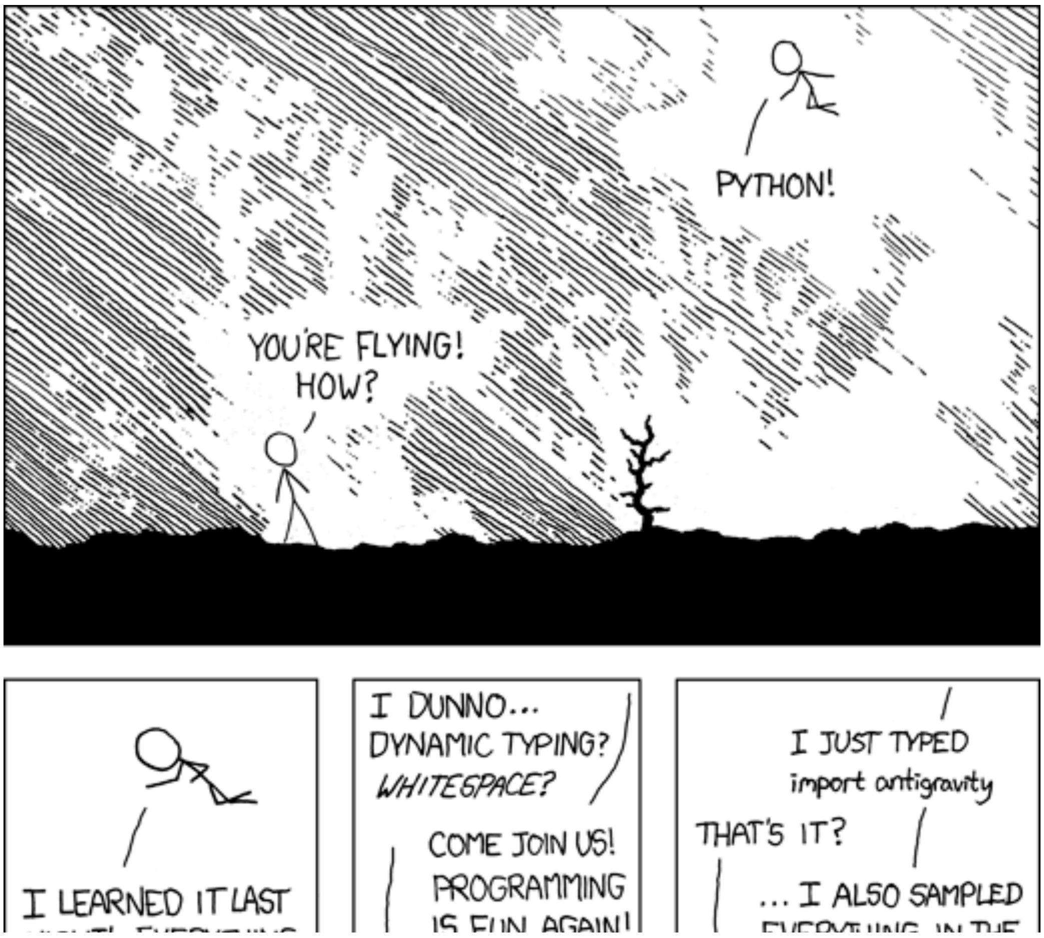
```
hello
world
```

I can `import` `a` and use `a.f()` when I want to call a function:

In [6]:
```
import a
a.f()
a.g()
```

```
hello
world
```

Python is a " batteries included " language. Which means it has a lot of libraries with useful functions . We can import these functions with the import syntax


PYTHON!
YOU'RE FLYING! HOW?
I LEARNED IT LAST
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN!
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED

The `random` library

```
In [46]: import random
```

```
In [9]: random.random() # Random numbers from 0 to 1

Out[9]: 0.2451302508918628
```

```
In [10]: random.randint(1,10) # Random integers from 1 to 10

Out[10]: 3
```

```
In [47]: random.choice(['London', 'Amsterdan', 'Berlin']) # Choose one at random

Out[47]: 'Amsterdan'
```

```
In [48]: random.sample(['London', 'Amsterdan', 'Berlin'],2) # Choose two at random

Out[48]: ['London', 'Berlin']
```

Running programs from the command line

So far we have seen how to make incredible things on Jupyter. Jupyter has recently emerged as a means of writing code but also as a method of sharing our programs. Jupyter is **not** the norm for creating programs with python but rather the exception. Most programs are written in `.py` files, for example `program.py` . We `run` these programs from the command line with the command:

```
python program.py
```

But what is the command line?

Each computer, regardless of operating system (windows, osx, Linux) has a program called the command line. When you open this program a console appears where the user can type commands and by pressing the Enter button, the computer executes them (or otherwise "runs" them). It may sound strange today but this is also the main way with which most programs run on your computer. When you double-click to open a program in a graphical environment, the computer internally simply executes the corresponding program command.

Consider the following: Graphic environments (eg. windows) have been around since the 1990s. How did people work on computers before? Answer: By writing commands on the command line.

Consider the following: Jupyter has been around since 2014 (and IPython, on which it has been based since 2001). RStudio has been around since 2010. How did people work in these popular languages before? Answer: With the command line.

Consider this: A program that requires a supercomputer to run on a remote machine and takes months to complete, how do we submit it for execution? We certainly can not have a jupyter open for months! The answer is we submit it through the command line.

Consider the following: In orer for a program to run requires the user to provide a number of input parameters. For example the location of the file containing my measurements. How do we "instruct" the program to use "this" file? Answer: With the command line.

The command line is the "classic" and timeless interface through which we interact with the computer. Computers and operating systems are built with this philosophy: The user will send me commands and I will execute them. Graphics systems came after and work "above" this logic.

Then why do we work with jupyter?

Using the command line requires some knowledge and experience which often acts as a barrier to those who want to learn programming "now!". Especially in the sciences where programming is auxiliary (biology, chemistry, engineering) students find the command line as an extra obstacle.

Jupyter (and notebooks in general), came to replace the command line with the "cell" logic we have already seen. In this way students can directly practice programming in a familiar environment such as the browser. But as we have seen, jupyter provides a subset of the abilities of a modern programming environment. If we want to "unlock" all the magic of programming we have to get to know the command line.

How do we start with the command line?

There are many tutorials online. For example: https://www.youtube.com/watch?v=MBBWVgE0ewk&list=PL6gx4Cwl9DGDV6SnbINlVUd0o2xT4JbMu

It is very important to follow a similar tutorial and get acquainted with the command line.

In these tutorials you will also learn a lot about how computers work and how different files are organized.

Those who have python anaconda can write `anaconda prompt` in the windows search in the start menu. See here https://www.youtube.com/watch?v=dgjEUcccRwM how it is done at 1:45. Doing so will open a command line interface where python will be "set up" and configured correctly. If you have a problem you can send me an email

How do we write a program so as to run it from the command line?

To do this, you need to install a text editor for programs. **Caution!** Microsoft word is not suitable for this! Not even the notepad or wordand (with a lot of effort you might succeed in notepad). You need a special program that helps the user to write python code. Some options are Notepad++ and Sublime. Through any of these programs try to create a file named: `program.py` which has the following content:

```
print ("Hello World!")
```

Where do I save this file?

The file should be saved in the same directory from which you run the `python program.py` command. You can see which directory you are in through the command line. The directory you are right now is usually shown on the command line as: `C:\Users\Alex` or something. If you type `cd` and press enter this directory will also appear. Next you need to save the `program.py` file to this directory from the text processor. Finally you can "run" the code in this file with:

```
python program.py
```

If no error occurs, you will see the text: `Hello World!` In the command line interface.

What is this strange `>>>` ?

If instead of `python program.my` you type `python` and then press enter, you will enter the python command line program which is different from that of your computer. In this environment you just type python. To "exit" from this press `exit()` .

Writing programs for the command line

Creating command line programs is not much different from jupyter. The main difference is that the command in programs are executed in the order in which they are written. While in jupyter a cell below can be executed before a cell above. For example: first I define a two cells below and then I print a cell below:

```
In [13]: print (a)
```

```
3
```

```
In [12]: a=3
```

In contrast, the following program will not run on the command line:

```
In [4]:  %%writefile program.py

         print (a)
         a=3
```

```
Writing program.py
```

```
In [5]:  !python program.py
```

```
Traceback (most recent call last):
  File "program.py", line 2, in <module>
    print (a)
NameError: name 'a' is not defined
```

The variable ___name___

We saw above how to build our own library and how to import the functions and variables it contains (with the `import` syntax). But here we also showed how to make a program that we run from the command line.

Which of these two methods is more appropriate for sharing our code? Or else, if we have to "give" our code somewhere, in what form will we give it? As a library that will be imported or as a program that will run from the command line?

The answer is: It's your choice! either as a library or from the command line or with both methods. Let's explore this 3rd option a bit.

For starters let's build a simple library:

```
In [14]:  %%writefile program.py

          def amazing_function(x):
              return x*2
```

```
Writing program.py
```

And let's import it:

```
In [15]:  from program import amazing_function
          amazing_function(10)
```

```
Out[15]:  20
```

We notice that if we "run" the `program.py` program nothing happens, since we do not print anything in program.py anywhere:

```
In [8]:  !python program.py
```

Is it possible to know if someone called me from a library or directly from the command line? First of all, why is this important? When we give a program to someone else (or just when we have it free on the Internet), we want to enable other users to use it in the most possible ways. Some developers will prefer to use our code as a library and some will run it from the command line.

When they run our code as a library (via `import`) then they "pass" the parameters directly to our functions. For example before we passed the value 10 to the `amazing_function` function. But how do we pass values to functions when we run a program from the command line? Let's explore this!

For starters, let's look at what the special variable `__name__` contains:

In [16]:
```
%%writefile program.py

def amazing_function(x):
    return x*2

print (__name__)
```

```
Overwriting program.py
```

What will print if we `import` the `program.py` ?

In [1]:
```
import program
```

```
program
```

This variable contains the name of the library to which it belongs! Wait a moment, when I run it from the command line there is no library, what will it print then?

In [5]:
```
!python program.py
```

```
__main__
```

Ir printed `__main__` ! Or else when a program is loaded via import then `__name__` contains the name of the library that contains it. But when we call it from the command line, then it contains the value: `___main__` .

ok, and how can I use this?

I can write code that runs only when the program is run from the command line and not as an import and vice versa:

In [19]:
```
%%writefile program.py

def amazing_function(x):
    return x*2

if __name__ == '__main__':
    print ('I was called from the command line!')
else:
    print ('I was loaded from an import statement')
```

```
Overwriting program.py
```

In [1]:
```
import program
```

```
I was loaded from an import statement
```

In [2]:
```
!python program.py
```

```
I was called from the command line!
```

Nice!, we said before that when we load a code throught `import` then the programmer who imported the code has the responsibility of passing the parameters to the imported functions.

But if we do not import the code and use instead the command line, how do we pass parameters to these functions?

Passing parameters from the command line

Passing parameters from the command line into the programs that are called from it is one of the smartest ideas but also the basic function of the command line. If you think about it you will see that you are already doing this when you write: `python program.py`. When the command line "sees" `python program.py`, it breaks it into words. The first word is the program that it will call. All other words after the first are parameters that will be passed to the program. In essence, what the comma line understands is: "Run the `python` program with the `program.py` parameter". The command line has no idea that program.py is a file. It considers it just a string which must be passed as a parameter to the python program.

To pass parameters to your program, all you have to do is add these parameters to the command line. For example:

In [2]:
```
!python program.py 10
```

```
I was called from the command line!
```

Nice, but how do we read the value "10" through python? There are 2 ways to do this. The first is by using the `sys` library which contains the list of `args` (arguments) with all the parameters from the command line:

In [3]:
```
%%writefile program.py
import sys

def amazing_function(x):
    return x*2

if __name__ == '__main__':
    print (sys.argv)
```

```
Overwriting program.py
```

In [4]:
```
!python program.py 10
```

```
['program.py', '10']
```

We notice that:

- the parameters we "passed" to our program in the form of a list.
- All parameters passed as a string. The command line does not distinguish between different types of variables. All parameters are strings.
- The first parameter is `program.py`. This is obvious. Remember that we said that the first word is the program (`python`) and the rest are the parameters that we pass in the program.

Now we can do whatever we want with this list. For example:

In [5]:
```
%%writefile program.py
import sys

def amazing_function(x):
    return x*2

if __name__ == '__main__':
    parameter = int(sys.argv[1])
    print (amazing_function(parameter))
```

```
Overwriting program.py
```

In [6]:
```
!python program.py 10
```

```
20
```

Also our program runs with both different ways!

In [7]:
```
from program import amazing_function

print (amazing_function(10))
```

```
20
```

Modern programs mainly in genetics and bioinformatics have a huge number of parameters. To be precise, these parameters offer tremendous expressiveness. Using the command line you can perform complex analyzes without having to write a single programming line. For example, see here a summary (!) of the parameters that samtools accepts as a command line program for managing files derived from sequencing experiments.Other examples are GATK and plink2 with over 100 parameters each.

If your program gets at most 3 parameters `sys.argv` might be enough. For more parameters the list provided by `sys.argv` is not enough. For this reason python has the argparse library to help you "parse" the parameters from the command line and pass it "painlessly" to your program.

I suggest you read the very good presentation and analysis of argparse by realpython .

The following is an example. We declare two parameters, one mandatory and one optional:

In [8]:
```
%%writefile program.py
import argparse

def amazing_function(x):
    return x*2

if __name__ == '__main__':

    parser = argparse.ArgumentParser(description="Amazing program")
    parser.add_argument('value', help="Number of desired iterations")    #
    parser.add_argument('--par', help="Value of the converge cutoff")    #

    args = parser.parse_args()

    value = int(args.value)
    par = args.par

    print (amazing_function(value))
    if not par is None:
        print ('The value of par={}'.format(par))
```

```
Overwriting program.py
```

Calling with the `-h` parameter our program displays a help message on how to use it:

In [11]:
```
!python program.py -h
```

```
usage: program.py [-h] [--par PAR] value

Amazing program

positional arguments:
  value       Number of desired iterations

optional arguments:
  -h, --help  show this help message and exit
  --par PAR   Value of the converge cutoff
```

Let's call it:

In [9]:
```
# If we don't use any parameters then an error will appear
# This is becase we declared at least one mandatory parameter
!python program.py
```

```
usage: program.py [-h] [--par PAR] value
program.py: error: the following arguments are required: value
```

In [10]:
```
# Let's add this parametera
!python program.py 10
```

```
20
```

In [11]:
```
# Let's add the mandatory and the optional parameter
!python program.py 10 --par hello
```

```
20
The value of par=hello
```

I repeat that all the code that parses and processes the parameters is inactive if we import the code through the `import` statement:

In [12]:
```
from program import amazing_function
```

Exceptions

You might have noticed that when something goes wrong python sends a message:

```
In [22]: a=1/0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-22-023a503edd86> in <module>
----> 1 a=1/0

ZeroDivisionError: division by zero
```

All errors that can occur in python belong to a new category of objects: exceptions. An exception when it happens TERMINATES the program:

```
In [13]: a=1/0
         print ("hello")
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-13-310b257491c9> in <module>
----> 1 a=1/0
      2 print ("hello")

ZeroDivisionError: division by zero
```

Notice that the program never printed "hello" because it simply "crashed" or otherwise ended unexpectedly when an error occurred (division by 0)

How can we check if an exception has happenend and how can we continue running even when an exception has occurred? We can run the code snippet inside a try..except block:

```
In [25]: try:
             a=1/0
         except:
             print ("Something bad happened")
```

```
Something bad happened
```

Commands after the point where the exception was made do NOT RUN!

```
In [14]: try:
             a=1/0
             print ("This will never run..")
         except:
             print ("Something bad happened")
```

```
Something bad happened
```

However, the commands after the try..except block, run normally:

In [15]:
```
try:
    a=1/0
    print ("This will never run..")
except:
    print ("Something bad happened")

print ("This will run!")
```

```
Something bad happened
This will run!
```

There are many types of exceptions. Here is a list of all of them . Instead of using try..except we can choose exactly which exception we want to catch:

In [16]:
```
try:
    a=1/0
except ZeroDivisionError:
    print ('Division with 0')
```

```
Division with 0
```

We can thus "catch" many kinds of errors:

In [17]:
```
divisor = 0
try:
    a=1/divisor
    open('I_DO_NOT_EXIST.txt')
except ZeroDivisionError:
    print ('Division with 0')
except FileNotFoundError:
    print ('File not found')
except:
    print ('Something else bad happened')
```

```
Division with 0
```

In [19]:
```
divisor = 1
try:
    a=1/divisor
    open('I_DO_NOT_EXIST.txt')
except ZeroDivisionError:
    print ('Division with 0')
except FileNotFoundError:
    print ('File not found')
except:
    print ('Something else bad happened')
```

```
File not found
```

In [20]:
```
divisor = 1
b = [1,2]
try:
    a=1/divisor
    b[3] = 5
    open('I_DO_NOT_EXIST.txt')
except ZeroDivisionError:
    print ('Division with 0')
except FileNotFoundError:
    print ('File not found')
except:
    print ('Something else bad happened')
```

```
Something else bad happened
```

If we use:

```
try:
    ...
except:
    ...
```

or

```
try:
    ...
except Exception
    ....
```

Then we have caught ALL possible exceptions.

We can store in a variable the exception that was caused by a command and get more information about the error that caused it:

In [21]:
```
try:
    a=1/0
except Exception as e:
    print ('This error happened: {}'.format(e))
```

```
This error happened: division by zero
```

We can catch many types of exceptions and store it in one variable:

In [35]:
```
divisor = 1
try:
    a=1/divisor
    open('I_DO_NOT_EXIST.txt')
except (ZeroDivisionError, FileNotFoundError) as e:
    print ('This just happened: {}'.format(e))
```

```
This just happened: [Errno 2] No such file or directory: 'I_DO_NOT_EXIST.tx
t'
```

Also instead of try..except we can use try..except..else. The code inside the `else` clause is executed only if an exception has NOT occurred.

In [36]:
```
divisor=0
try:
    a=1/divisor
except ZeroDivisionError:
    print ('Division failed..')
else:
    print ('Division succeeded!!')
```

```
Division failed..
```

In [37]:
```
divisor=1
try:
    a=1/divisor
except ZeroDivisionError:
    print ('Division failed..')
else:
    print ('Division succeeded!!')
```

```
Division succeeded!!
```

We can also use try..except..finally. The code inside `finally` runs always regardless if an exception has happened or not:

In [22]:
```python
divisor=0
try:
    a=1/divisor
except ZeroDivisionError:
    print ('Division failed..')
finally:
    print ('I always run')
```

```
Division failed..
I always run
```

In [23]:
```python
divisor=1
try:
    a=1/divisor
except ZeroDivisionError:
    print ('Division failed..')
finally:
    print ('I always run')
```

```
I always run
```

Then what is the point of having a `finally` clause? If an exception occurs the code will finally run **after** the exception. This is very useful when we want to "clean-up" the.. mess that we created by trying to run the code. E.g. to close the files that we opened. For example:

In [24]:
```python
def save_weird_calculation_to_file(file, a,b):
    value = a/b
    file.write("{}\n".format(value))

f = open("amazing_results.txt", 'w')
a=4
b=1
try:
    save_weird_calculation_to_file(f, a, b)
finally:
    print ('Closing file!')
    f.close()
```

```
Closing file!
```

In the above we notice that no error occurred and the file was closed normally. Now let's set the value 0 to b:

In [25]:
```python
def save_weird_calculation_to_file(file, a,b):
    value = a/b
    file.write("{}\n".format(value))

f = open("amazing_results.txt", 'w')
a=4
b=0
try:
    save_weird_calculation_to_file(f, a, b)
finally:
    print ('Closing file!')
    f.close()
```

```
Closing file!
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-25-287591b9dc58> in <module>
      7 b=0
      8 try:
----> 9     save_weird_calculation_to_file(f, a, b)
     10 finally:
     11     print ('Closing file!')

<ipython-input-25-287591b9dc58> in save_weird_calculation_to_file(file, a, b)
      1 def save_weird_calculation_to_file(file, a,b):
----> 2     value = a/b
      3     file.write("{}\n".format(value))
      4
      5 f = open("amazing_results.txt", 'w')

ZeroDivisionError: division by zero
```

We observe the following: The error happend, it threw an exception and nevertheless it closed the file. If we had not used the try..finally the file would not have been closed.

An example using try..except..else..finally:

In [26]:
```python
divisor = 0
try:
    a=1/divisor
except ZeroDivisionError:
    print ('Division failed')
else:
    print ('Division succeeded')
finally:
    print ('I always run')
```

```
Division failed
I always run
```

CAUTION! The fact that we have used try..except..else..finally etc does not mean that we have "covered" all possible exceptions:

In [45]:
```python
try:
    mpaklavas
except ZeroDivisionError:
    print ('Division with zero')
else:
    print ('No exception happened')
finally:
    print ('I always run')
```

```
I always run
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-45-0457af044c52> in <module>
      1 try:
----> 2     mpaklavas
      3 except ZeroDivisionError:
      4     print ('Division with zero')
      5 else:

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

In the above example a `NameError` exception occured which we did not catch anywhere.

We can "throw" or "raise" and our own exception:

In [27]:
```
raise Exception('something is wrong with left phalange')
# https://www.youtube.com/watch?v=DrwVB4vMx-Q&feature=youtu.be&t=30
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-27-b47ac901eec8> in <module>
----> 1 raise Exception('something is wrong with left phalange')
      2 # https://www.youtube.com/watch?v=DrwVB4vMx-Q&feature=youtu.be&t=30

Exception: something is wrong with left phalange
```

In [28]:
```
def print_my_name(name):
    if name in ['hell', 'her', 'Jane', 'Stacey', 'quiet']:
        raise NameError("That's not my name") # https://www.youtube.com/wa

    print ('My name is:', name)
```

In [29]:
```
print_my_name('Alex')
```

```
My name is: Alex
```

In [30]:
```
print_my_name('Stacey')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-30-5347c2f26f4c> in <module>
----> 1 print_my_name('Stacey')

<ipython-input-28-bc05d5512eaa> in print_my_name(name)
      1 def print_my_name(name):
      2     if name in ['hell', 'her', 'Jane', 'Stacey', 'quiet']:
----> 3         raise NameError("That's not my name") # https://www.youtub
e.com/watch?v=v1c2OfAzDTI
      4 
      5     print ('My name is:', name)

NameError: That's not my name
```

The novelty of exceptions is that we can "catch" them not only exactly where they happened but also outside the function that caused them:

In [31]:
```
def f():
    raise Exception('This is a mistake!')

def g():
    f()

def h():
    g()

try:
    h() # And over here there might be a small exception
except Exception as e:
    print ('Error: {}'.format(e))
```

```
Error: This is a mistake!
```

Advanced: We can make our own exception:

In [32]:
```
# We will later explain what classes are
class MyFabulousException(Exception):
    pass
```

```
In [33]: raise MyFabulousException
```

```
---------------------------------------------------------------------------
MyFabulousException                       Traceback (most recent call last)
<ipython-input-33-cb966f875a08> in <module>
----> 1 raise MyFabulousException

MyFabulousException:
```

Collections

The collections library contains some additional structures (besides lists, dictionaries and sets) for data management. Two of the most important are Counter and defaultdict. With `Counter` we can count the occurences of the unique items of any structure:

```
In [34]: from collections import Counter
```

```
In [35]: Counter('hello world')
```

```
Out[35]: Counter({'h': 1, 'e': 1, 'l': 3, 'o': 2, ' ': 1, 'w': 1, 'r': 1, 'd': 1})
```

```
In [36]: Counter([5,4,3,5,6,7,6,5,6,4,3,2,1,2,8])
```

```
Out[36]: Counter({5: 3, 4: 2, 3: 2, 6: 3, 7: 1, 2: 2, 1: 1, 8: 1})
```

The most common element:

```
In [37]: a = Counter('helloworld')
         a.most_common(1)[0][0]
```

```
Out[37]: 'l'
```

We can add two Counters together:

```
In [38]: a = Counter('Mitsos')
         b = Counter('Kostas')
         print (a+b)
```

```
Counter({'s': 4, 't': 2, 'o': 2, 'M': 1, 'i': 1, 'K': 1, 'a': 1})
```

Defaultdict is a dictionary in which we can set a default value:

```
In [39]: from collections import defaultdict

         d = {}

         a = defaultdict(int) # Use 0 as a default value
```

```
In [40]: print (a['mitsos'])
```

```
0
```

```
In [41]: print (d['mitsos']) # This raises an error
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-41-d4501b84c28d> in <module>
----> 1 print (d['mitsos']) # This raises an error

KeyError: 'mitsos'
```

```
In [42]: a = defaultdict(list)

         print (a['mitsos'])
```

```
[]
```

Where is this useful? Sometimes in an iteration, we want to do an operation with the elements of each iteration in a dictionary. Normally, each time, we have to check if the key is in the dictionary and if it is not there to create it. We can avoid this with defaultdict. For example, let a list of strings. Create a dictionary where each key be an integer. The values will be a list of strings that have length equal to the key.

```
In [43]: l = '''ἄνδρα μοι ἔννεπε, Μοῦσα, πολύτροπον, ὃς μάλα πολλὰ
         πλάγχθη, ἐπεὶ Τροίης ἱερὸν πτολίεθρον ἔπερσε·
         πολλῶν δ᾽ ἀνθρώπων ἴδεν ἄστεα καὶ νόον ἔγνω,
         πολλὰ δ᾽ ὅ γ᾽ ἐν πόντῳ πάθεν ἄλγεα ὃν κατὰ θυμόν,
         ἀρνύμενος ἥν τε ψυχὴν καὶ νόστον ἑταίρων.
         ἀλλ᾽ οὐδ᾽ ὣς ἑτάρους ἐρρύσατο, ἱέμενός περ·
         αὐτῶν γὰρ σφετέρῃσιν ἀτασθαλίῃσιν ὄλοντο,
         νήπιοι, οἳ κατὰ βοῦς Ὑπερίονος Ἠελίοιο
         ἤσθιον· αὐτὰρ ὁ τοῖσιν ἀφείλετο νόστιμον ἦμαρ.
         τῶν ἁμόθεν γε, θεά, θύγατερ Διός, εἰπὲ καὶ ἡμῖν.'''.split()
```

Without the defaultdict:

```
In [44]: d = {}
         for word in l:
             length = len(word)
             if not length in d:
                 d[length] = []

             d[length].append(word)

         print (d)
```

```
{5: ['ἄνδρα', 'πολλὰ', 'ἱερὸν', 'ἄστεα', 'ἔγνω,', 'πολλὰ', 'πόντῳ', 'πάθεν
', 'ἄλγεα', 'ψυχὴν', 'αὐτῶν', 'αὐτὰρ', 'ἦμαρ.', 'Διός,', 'ἡμῖν.'], 3: ['μοι
', 'καὶ', 'καὶ', 'γὰρ', 'τῶν', 'γε,', 'καὶ'], 7: ['ἔννεπε,', 'ἔπερσε·', 'ἑτ
άρους', 'ἱέμενός', 'ὄλοντο,', 'νήπιοι,', 'Ἠελίοιο', 'ἤσθιον·', 'θύγατερ'],
6: ['Μοῦσα,', 'Τροίης', 'πολλῶν', 'θυμόν,', 'νόστον', 'τοῖσιν', 'ἁμόθεν'],
11: ['πολύτροπον,'], 2: ['ὃς', 'δ᾽', 'δ᾽', 'γ᾽', 'ἐν', 'ὃν', 'ἥν', 'τε', 'ὣ
ς', 'οἳ'], 4: ['μάλα', 'ἐπεὶ', 'ἴδεν', 'νόον', 'κατὰ', 'ἀλλ᾽', 'οὐδ᾽', 'περ
·', 'κατὰ', 'βοῦς', 'θεά,', 'εἰπὲ'], 8: ['πλάγχθη,', 'ἀνθρώπων', 'ἑταίρω
ν.', 'ἀφείλετο', 'νόστιμον'], 10: ['πτολίεθρον', 'σφετέρῃσιν'], 1: ['ὅ', 'ὁ
'], 9: ['ἀρνύμενος', 'ἐρρύσατο,', 'Ὑπερίονος'], 12: ['ἀτασθαλίῃσιν']}
```

With defaultdict:

```
In [84]: d = defaultdict(list)
         for word in l:
             d[len(word)].append(word)
         print(d)
```

```
defaultdict(<class 'list'>, {5: ['ἄνδρα', 'πολλὰ', 'ἱερὸν', 'ἄστεα', 'ἔγν
ω,', 'πολλὰ', 'πόντῳ', 'πάθεν', 'ἄλγεα', 'ψυχὴν', 'αὐτῶν', 'αὐτὰρ', 'ἦμα
ρ.', 'Διός,', 'ἡμῖν.'], 3: ['μοι', 'καὶ', 'καὶ', 'γὰρ', 'τῶν', 'γε,', 'καὶ
'], 7: ['ἔννεπε,', 'ἔπερσε·', 'ἑτάρους', 'ἱέμενός', 'ὄλοντο,', 'νήπιοι,', '
Ἠελίοιο', 'ἤσθιον·', 'θύγατερ'], 6: ['Μοῦσα,', 'Τροίης', 'πολλῶν', 'θυμό
ν,', 'νόστον', 'τοῖσιν', 'ἁμόθεν'], 11: ['πολύτροπον,'], 2: ['ὃς', 'δ᾽', 'δ
᾽', 'γ᾽', 'ἐν', 'ὃν', 'ἥν', 'τε', 'ὣς', 'οἳ'], 4: ['μάλα', 'ἐπεὶ', 'ἴδεν',
'νόον', 'κατὰ', 'ἀλλ᾽', 'οὐδ᾽', 'περ·', 'κατὰ', 'βοῦς', 'θεά,', 'εἰπὲ'], 8:
['πλάγχθη,', 'ἀνθρώπων', 'ἑταίρων.', 'ἀφείλετο', 'νόστιμον'], 10: ['πτολίεθ
```

```
ρον', 'σφετέρῃσιν'], 1: ['ὅ', 'ὁ'], 9: ['ἀρνύμενος', 'ἐρρύσατο,', 'Ὑπερίονο
```

```
In [ ]:
```