ISEL \ DEETC

MEIM – MEIC

AMD / MLDM (Machine Learning and Data Mining**) – Module of Practice**

Paulo Trigo Silva

---

1. (*initial*) Python – (brief) exploration of data structures

Consider the file: “o01_oEssencialDaLinguagemPython.pdf”. It is written in Portuguese but we will only use specific programming examples (which are written in Python!). Note that it refers to v2.* of Python; from v2.* to v3.* there were some major changes that you will notice! for example “print” is in v3.* a function so you must call using brackets, e.g., `print( “Hello World!” )`.

a) Go to page #9 and execute the example; our “hello world” first Python program!

b) Go #15 to #21 and explore “list data structure”; one of Python’s most important data structure.

c) Go #22 to #26 and explore “dictionary (hash table) data structure”; also very important in Python.

d) Go #27 to #29 and explore “tuple data structure”; similar to a list but immutable.

e) Go #30 to #32 and explore “conversion of data structures” and “list comprehension writing”.

f) Go #33 to #34 and explore “file manipulation”.

g) About data structures, you may also explore the examples in “p01_aByteOfPython_v3.pdf”; go to “Data Structures” section and implement the examples.

2. (*initial*) Python – control flow, functions, modules and a problem

Consider the file: “p01_aByteOfPython_v3.pdf” (the file “p01_aByteOfPython_v2.pdf”, which describes Python version 2.*, is also available for historical/sentimental reasons!...)

a) Go to “Control Flow” section and explore the “if”, “while”, “for”, “break” and “continue” statements.

b) Go to “Functions” section and explore the function definition; formal and actual parameters and local variables. *Note*: it is “forbidden” to use the “global” statement in all your programming tasks of this subject.

c) Go to “Modules” section and explore the module definition; take special attention to the module attribute “`__name__`” and how it can be used to define an “entry point” for your Python program.

d) About modules, you may also explore the examples in “o01_oEssencialDaLinguagemPython.pdf”; go from #45 to #51 and implement the examples.

e) [***“backup problem”***] Go to “Problem Solving” section of file “p01_aByteOfPython_v3.pdf”. Consider the “backup problem” and implement the three versions that are described. But, instead of implementing each version as a separate module, implement each version as a different function (e.g., backup_v01, backup_v02, backup_v03) and define the formal parameters that are necessary to make the function reusable (e.g., the list of files and folders to backup, the destination folder, the path to the “zip” application, among others that you may consider). *Very important*: you must defined the application “entry point”; use the “`__name__`” module attribute.

Paulo Trigo Silva

3. (*middle*) Python – analyze and implement functional requirements

Consider the file: "a01_aFunction.py".

a) Analyze the code (in "a01_aFunction.py"). Make a change in a single line (only in one line) so that the algorithm only terminates in case the "`a_list`" contains an even number (along with the words 'exit' and 'quit'). Note that the algorithm, as it is, terminates whenever it reads a value that belongs to "`a_list`". *Hint*: consider the list comprehension technique in its complete syntax; recall that the function 'isinstance(`var`, `type`)' returns True, or False, whether `var` is, is not, of `type` (`str` and `int` refer to types *string* e *integer*).

b) Add a single line to the code so that we only have: `if v.upper() in lista: break`; i.e., eliminate the remaining conditions but retian the possibility of terminating with 'exit' or 'quit'.

4. (*middle*) Python – module interaction and module rewriting

Consider the file: "a02_anotherBehavior.py".

a) Analyze the code (in "a02_anotherBehavior.py"). Execute and explain the behavior that you get.

b) Analyze the "`lineList`" data structure. Iterate the "`lineList`" and print each line separately. *Hint*: recall that "`print x`" adds a newline and "`print x,`" does not add a newline.

c) Remove the comments from the label "`ITEM-`**`A`**" until the line before "`ITEM-`**`B`**" label (in IDLE, to remove comments select set group of lines and go "Format \ Uncomment Region"). Execute the program and analyze the result. Undo (manually) the effect of executing the program.

d) Remove the comments from the label "`ITEM-`**`A`**" until the line before "`ITEM-`**`C`**" label. Execute the program and explain the result that you get.

e) Initiate a "command prompt" window (i.e., a *DOS shell*) and execute again, in that window, the previous program; explain the behavior that you get.

f) Undo (manually) the collateral effects of all the above executions.

5. (*middle*) Python – dynamic evaluation of a module

Consider the file: "a02_anotherBehavior.py".

a) Remove the comments from the label "`ITEM-`**`A`**" until the line before "`ITEM-`**`D`**" label. Execute the program and explain the result that you get. Undo (manually) the effect of executing the program

b) Go to "a01_aFunction.py" and edit a new function named "`average( aList )`" that returns the average of all numeric values contained in the "`aList`". Execute and test to guarantee that your implementation is correct.

c) Remove the comments from the label "`ITEM-`**`A`**" until the line before "**`THE-END`**" label. Execute the program and explain the result. Change the code in order to execute your "`average`" function.

Paulo Trigo Silva

6. (*advanced*) Python – implement "command-line" design pattern

a) Consider the file: "e_Config.py "

b) Analyze the code and execute the code.

c) Uncomment the last lines and execute the test code.

d) Change the command-line parameters; instead of "-a", "-b", "-c", define "-x", "-y", "-z", "-t" with some default values.

e) Recall the "***backup problem***" (cf., exercise 2) and apply thos "command-line" design pattern in order to launch the backup from the command line with the parameters provided by the user.

7. (*advanced*) Python – experiments with "decorator" design pattern

Consider the file: "u_function_of_function_and_decorator.py "

a) Analyze the code and execute the program.

b) This is just a theoretical exercise to illustrate that the notion of class can be formulated as a construction of functional composition (which derives from the lambda function calculus). So, as an exercise, follow the class formulation (i.e., `def c( v=0 )` definition) and add a new method.

c) You may also want to extend the class formulation and incorporate the meta-class concept. To experiment on that you just need to define another function (e.g., named "`meta_class`") over "`c`" function that returns either the "`c`" function or another function that follows the same formulation as the "`c`" function. That is, the meta-class will be a class and its methods will be the classes.

d) Explore the examples with the @decorator label. Change the order @f, @g into @g, @f and try also @f, @f, @f, @g and some other variations of yours.

8. (*advanced*) Python – "singleton" (and "decorator") design pattern

Files: "x_singleton.py", "z01_singleton_usage.py" and "z02_singleton_usage_anotherContext.py"

a) Analyze the code in "z01_singleton_usage.py" and execute the program.

b) In order to better understand the singleton mechanism, go to "x_singleton.py" and uncomment the "print" statement. Then, you will see the exact moment when the decorator is applied.

c) Analyze the code in "z02_singleton_usage_anotherContext.py" and execute the program. Here you see the singleton being invoked from a different context (module) from the one it was defined.