Prof. Dr. A-S. Smith
am Lehrstuhl für Theoretische Physik I
Department für Physik
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Advanced Python for Research Projects

Exercise sheet 1

Problem 1.1 *Creating a standalone python script*

To set up for future tasks, we want to create a simple python script that can be run from the command line.

a) Create a file called "basic_setup.py" and add a shebang to make the file executable from the command line without specifying the python interpreter.

b) Use the `argparse` package (see `https://docs.python.org/3/library/argparse.html` for documentation, you need to import it first) to allow for two options of the script to be set:

1. First, add an option to set the log level via the `--log` option. The log level should be a string and either `warn`, `info`, `info`, `debug`, or `error`. Store the resulting option in a variable `loglevel`. The default setting should be `warn`.
2. Add an option `-i, --input_folder`, that allows the user to specify an input folder for future processing. The default option should be the current directory. Store the value in a `input_folder` variable.
3. Add a help text to the argparse setup and the individual arguments. The help text for the script should state that it is a script to display the files and folders in a directory in alphabetic order. The help text should specify which values can be set on each argument and which default values will be applied if none are set.

c) Test that your script is working as intended by running `./basic_setup.py -h` or `./basic_setup.py --help`, which should display the help text associated with your script. This should contain all of the description text you configured with the argparse package. If the help text does not contain all of the arguments, go back and fix your code to make sure, all of your help text is displayed correctly.

d) To finish up this initial setup, add a default output using the `print` method to state which log level has been set and which directory will be analyzed.

Problem 1.2 *Setting up a logger*

Now, we want to set up a logger using the `logging` package as presented in the lecture (see `https://docs.python.org/3/library/logging.html` for documentation). We want to create the logger for our main script using the `__name__` variable as the logger name.

a) Expand the script created in the prior exercise problem and create a Logger object in a `logger` variable for which you set the loglevel according to the `loglevel` variable configurable via the command line options. Also, depending on your configuration of the argparse package: If the log level is none of the permitted options (may not be possible depending on your setup of the arguments), output a warning via the logger, that the log level was not appropriately set and that the default option will be set instead.

b) Extend the logger with two handlers:

1. One writing to the command line (`StreamHandler`)

2. One writing all output to a log file in the current directory (`FileHandler`).

c) Modify the existing code so that the information about the set log level and the chosen directory are only displayed if the log level is `info` or `debug`. Use the `logger.info` function for this.

d) We want to catch some usual issues first and also add appropriate logging outputs for those occasions. Use the `os.path.isdir()` and `os.path.exists()` functions to check whether the provided input folder path exists and is also a directory. If not, log an error message stating that the provided path is not a directory.

e) We now want to store the log file in the chosen directory instead of the current folder. Modify your code to add the `FileHandler` only after the check whether the directory exists has been successful. The target log file should be called `.basic.log` (with a leading dot, to mark the file as hidden) and be within the input directory.

Problem 1.3 *Type checking*

We want to write our directory list in an extensible and reusable manner. For this purpose, we want to create a function `create_directory_listing`, with appropriate type hints.

a) The method should take two options: the directory path and the logger.

b) Use the `typing` package to add type hints to the function.

- The directory path should be provided as a `pathlib.Path` object. Make sure you import the necessary types and packages for this type hint to work.
- Find the appropriate type for the Logger from the logging package documentation and add the appropriate type hint to the logger parameter.
- The return type of the function should either be a pair of lists (`List`) of string objects (`str`) or a None object (if there is an error). Use the `Union` type from the typing package to implement this.

c) The check, whether the directory exists, should now be moved into the `create_directory_listing` function. Use the methods of the `pathlib.Path` object to check for existence and being a directory instead of the `os.path` module's functions.
Additionally, the setup of the `FileHandler` within the target directory should be moved into this newly created function. As the log level is not passed as a parameter to the function, you can get the currently set log level from the provided Logger and apply it to the `FileHandler` as well. Whenever an error or warning occurs, the issue should be logged with the logger provided to the function. If the error is unrecoverable like the directory not existing, return `None` from this function in accordance with the type hints. If the issue is recoverable, log a warning and continue with the function execution. Before returning from this function, remove the `FileHandler` from the logger again.

d) Using the `pathlib.Path` class's functionality, iterate over the entries of the directory. Use the `pathlib` module's functionality to check whether each entry is a directory or a file. Ignore everything that is neither a file or a directory, but add an `info` output for the logger that such an element has been encountered, including the respective name.

e) Also ignore all hidden entries of the directory, which are entries whose names start with a `"."`. However, you should log info messages for those entries, stating that hidden elements have been omitted.

f) Now, create two lists, one containing the names of all entries in the directory which are files, and one, which contains the entries which are directories (use the `.name` property of the `Path` object to get only the last part of the path). Sort each of these lists alphabetically. Return the results as a pair `(files, directories)`.

g) We now have a special case, when the directory exists but there are no entries in our return values, e.g. because all entries are hidden. Add a check whether both of our lists are empty before returning them and if that is the case, add a warning to the log that there are no visible entries in the chosen directory.

h) Now, the function is somewhat ambiguous: The user would not know, which of the lists in the return value is the list of files and which the list of directories. Add a docstring comment to document the right order to your function. Such a documentation document is a comment immediately following the ":" after the function name, arguments and return type declaration. It starts with `"""` (all three double quotes) and is terminated by the same three double quotes. In between, add a message explaining the functionality of the whole function including the possible error of the directory not existing, the logger being extended by the function, the order of entries in the return value and the return value in case of an error.

i) To tie up the module, use the function `create_directory_listing` in your module's main code to get the list of files and directories in the chosen path from the script's arguments. If None is returned, you can output an error message using `print` in addition to the log message explaining the error and terminate the program using `sys.exit(1)`, where the value 1 denotes an error. If the list of files and directories could be generated, write to the output a list of all file names first, where you start with a line stating "FILES:" followed by one line for each individual file name. Then follow it up with a line "FOLDERS:" followed by one line per directory/folder contained within the chosen directory. To signify success, either just let the program reach the end of your script or run `sys.exit(0)` (where 0 is the success flag for a program's execution) at the end of your code

j) We want to test the correctness of our code. Run `basic_setup.py -log info`" and compare the results to running `ls -ah`. Which differences do you notice? Would you expect there to be differences?

Problem 1.4 *Iterators*

We now want to create our own **iterator** to streamline the use of our `create_directory_listing` function. For this purpose, we want to create a class implementing the iterator interface (i.e., the `__iter__` and `__next__` methods), that calls the `create_directory_listing` function in its constructor and then provides the directories and files as iterator items.

a) Create a class `DirectoryIterator` with a constructor, `__iter__` and `__next__` methods. The constructor is supposed to accept three arguments:

   - the path to the directory to be analyzed,
   - a bool value `is_files`, which if set to True makes the iterator only return the filenames and if set to False makes the iterator only return directory names,
   - and lastly the logger to be used for logging purposes.

   In the constructor, set instance variables containing the path to the directory, the `is_files` value and a variable containing the `logger` provided to the constructor. Make sure these are not accidentally stored neither in global variables nor in class variables instead.
   Additionally, create an instance variable `entry_list`, in which you store either the list of files or the list of directories you obtain by calling the `create_directory_listing` function with the appropriate parameters. Choose, which to pick for your `entry_list` based on the `is_files` value.
   Think about how to best deal with errors or issues that may occur during this constructor and use logging appropriately. Setting the `entry_list` value to an otherwise impossible value may be a good way to signal an error during construction.

b) Implement a `__repr__` functionality for your class to help with debugging. As said in the lecture, it should tell you about all the values necessary for recreating this instance of the

`DirectoryIterator`, i.e. all parameters of the constructor. For the logger, you can call `repr(logger)` to obtain a representation of the logger.
When done, verify the result of your debug output, by outputting a test instance of the `DirectoryIterator` class. Again, use the variables of the `self` instance, no global or class variables.

c) Implement the `__iter__` method. (Hint: since our object implements the `__next__` method, just return the instance itself)

d) Implement the `__next__` method. To remember the index of the last entry you have returned, add a `entry_index` variable to your object/instance in the constructor, initialize it appropriately in the constructor to an invalid position, reset it in the `__iter__` method and update it upon every call to the `__next__` method. Check that you do not accidentally exceed the number of entries in your object's `entry_list` and raise a `StopIteration` exception when the last entry has been returned. (Note: You may choose to implement the `len()` functionality on your object to help with figuring out whether you are at the end and also allowing users to figure out, how many files or folders there are in your iteration)

e) We now verify that our `DirectoryIterator` class works as intended. Modify the main file code written in the previous exercise, where you used the return values of `create_directory_listing` to write the "FILES:" and "FOLDERS:" lists to instead use a for loop over a `DirectoryIterator` instance for each individual list. Verify, that the output of your program has not changed when switching to the `DirectoryIterator` instead of using `create_directory_listing` directly.

f) Finally, add sufficient documentation to your `DirectoryIterator` class and its methods. In the class definition, outline which purpose the class has. You should point out the meaning of the `is_files` option and that the class can be used as an iterator.
In the documentation to the constructor, explain the meaning of each indvidiual parameter and point out potential changes made to parameters. (Hint: think about the added FileHandler being added temporarily. If we did not remove it, this would be the place to point it out).
In the documentation for the `__iter__` method, mention that the iteration is reset to the first position when this function is called. Why would you mention that? Because it may impact your code if you maintain multiple variables containing the same object as an iterator that you would want to use independently. Also note that you cannot iterate over the same object at two points independently. Only after the first iteration is finished can you create a new iterator from the `DirectoryIterator` instance.
In the documentation to the `__next__` method, you do not necessarily have to mention that the `StopIteration` exception is thrown upon reaching the end of the entry list as it is part of the default contract for the `__next__` function. However, it would be reasonable to mention that it follows convention in this regard and that its outputs depend on the choice of the `is_files` option.