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 2

Problem 2.1 *Working with advanced data processing functions*

We want you to get familiar with the `map` and `reduce` function to process data sets first. There are certain disadvantages to using them compared to list comprehensions and generators which we will illustrate further in this exercise. Eventually, you will build an improved version of both the map and the filter functions. In this context, we will also illustrate the use of so-called lambda functions to simplify the code.

a) Create a file called "map_reduce.py" and add a shebang to make the file executable from the command line without specifying the python interpreter. This time, we do not immediately require documentation or command line arguments to be set up. However, it may be helpful to add a helptext using argparse to it anyway to explain that it is a script running some sample calculations.

b) We first want to demonstrate the `map()` function (see `https://docs.python.org/3/library/functions.html#map` for documentation). The `map`-function is applied to a function and at least one iterable, i.e. one list, sequence, iterator, etc. For example, you could call `map(double, a)` to convert all of the entries of list `a` to a `double` value, by applying the conversion to each element individually. Please note, that if you provide more than one sequence after the initial function, your function will need to accept as many arguments as you have provided iterables. Also, your sequences must all be of the same length.
Another relevant aspect is that the `map` function performs lazy evaluation, meaning that it returns an iterator and the individual values are only calculated as needed.
Let us practice the application of the `map` function on a few simple examples. For the following problems, try solving them once with a named function and once with a `lambda`-expression (see `https://www.w3schools.com/python/python_lambda.asp` for examples).

1. Create a list `intlist` of arbitrary integers. Write code to triple all numbers in a given list of integers using `map()` and apply it to your list. Store the result in a variable `tmp`. What happens if you execute `print(tmp)`? How does the behavior change, if you execute `print(list(tmp))` instead? And what happens, if you print the `list`-conversion of `tmp` twice after each other?
Now we want to illustrate another issue. Create a second variable `tmp2`, in which you store the result you get by applying the same `map`-call as before, applied to your list `intlist`. Then append to `intlist` two more integers of your choice. Only then print out the result of the conversion of `tmp2` to a `list` like `print(list(tmp2))`. What to you observe? What would you instead expect to happen?
The issues you observe here are a consequence of the lazy evaluation of `map`. Only once you iterate over the result of `map` are the values calculated. Also, if the iterator has been used once, it will have reached its end, therefore you can only convert it to a list once. Otherwise, the second and further subsequent conversions will be empty, because the iterator will be at its end. Also, the delayed evaluation leads to the list entries appended to the list after the call to `map` to still be evaluated by the `map` conversion to list, because the list is only read when the `map`-results are truly required. If you want to fix this issue, you should

immediately convert the result of a `map`-call to a list and only operate on that list from that point on. Otherwise, you may suffer from the issues arising from lazy (i.e. only once the values are needed) evaluation.

2. Create a list of strings, where each string is a sentence (you can e.g. use sentences copied from this instruction sheet). Write a Python program to split each string within this list into a list of words (using space as a delimiter) using `map` and collect the results into a new list, where each entry is a list of words. (Hint: lookup the `.split()` method of strings) Output the initial list as well as the result to verify your code.
3. Create two different lists with floating point values. Write python code to add these two lists and code to find the element-wise difference between them. Use the `map()` function. Output the initial lists as well as the results to verify your code.
4. Create three different lists of integers. Write python code to add the three given lists using `map` (and `lambda`). Use the option to provide more than one sequence to the `map()`-function. Verify your code by printing the inputs and the results.

c) Let us now use the `reduce()`-function (see `functools.reduce()` documentation `https://docs.python.org/3/library/functools.html`). Its default signature is `reduce(func, iterable[, initial])`, where it is optional to provide an initial argument. The `func` function always receives two arguments, the first being the accumulated value so far, the second being the next entry of the list. If the initial value is provided, this means, that the first call will be roughly `func(initial, iterable[0])`. If no initial value is provided, the first call will be roughly `func(iterable[0], iterable[1])`. Let's do some simple calculations with reduce:

1. Create a list of integers. Use the `reduce` function to calculate the sum of all entries of this list (equivalent to a `sum()` function call). For a list `[1,5,2,3]` the code should e.g. calculate $((1 + 5) + 2) + 3$.
2. Use `reduce` to calculate "the first entry" minus "all other entries" of the list.
3. Use `reduce` to calculate the alternating sum of a list, e.g. for `[1,5,2,3]` this would be $((1 - 5) + 2) - 3$. (Hint, if you negate the cumulative argument and add the next number, you will get an alternating sum with a `+` as the last operation. You may then only need to negate the full result if the list has an even total length.)

d) Now we want to be a bit more advanced and combine `map` and `reduce` as well as introducing the `filter` function. The filter function takes a function accepting one argument and an iterable `filter(predicate, iterable)`. It returns an iterator that only contains the values x returned by `iterable` for which $predicate(x)$ is `True`. By combining these three functions, we can perform even more complex calculations quickly:

1. Create an initial list of integers. Use `map` and `reduce` to calculate the sum of the squares of all integers in this list. (Hint: First map to a list of all numbers squared and then reduce to calculate the sum)
2. Create an initial list of integers. Use `map`, `filter` and `reduce` to only add up the squares of numbers that are odd (i.e. not even/divisible by 2).
3. Create an initial list of integers. Use `map` to convert the integers to their string representations. Use `filter` to only retain those integers with an odd number of digits and then combine all remaining integers into a comma-separated string using `reduce`, where you use the `initial` argument of it to prefix the resulting string with "0".

e) Python has built-in generators and list comprehension, meaning that we can write statements like the following:

```
[ i*i for i in list(range(10)) if i% 2== 1]
```

This code returns a list (as expressed by the square brackets []). The inner expression has to be read very carefully to be understood correctly, so let us explain it in detail:

1. First, we denote the expression that will be calculated in each entry of the result: `i*i`
2. Then, the `for` statement explains, where our variables come from. Directly following the for is the list of variables. In our case `i`.
3. Then, the `in` statement tells us, which values of `i` we will take as an initial input. Here, the list is constructed from the range of numbers from 0 to 9. We would not need to convert the `range` object to a `list`, however, as the list comprehension automatically exhausts iterators.
4. Lastly, the `if` statement allows us to filter which values of `i` we want to retain. The expression following the `if` needs to evaluate to `True` for all entries, we do not want to discard.

In summary: The list comprehension provides eager evaluation (the result will be a list, not an iterator), we can calculate arbitrary functions in the initial expression (e.g. `i*i`, but we can use arbitrary function calls as well), we can use arbitrary input iterators and we can perform filtration on the go. Consequently, list comprehensions are a very python-esque way to do, what map and filter can do.
Go back to the previous problems and redo them with the operations previously done by `map` and `filter` with a list comprehension instead. If there is a reduce step, retain the `reduce` call. Validate the accuracy of your code, by comparing the results with and without list comprehension.

Problem 2.2 *Building our own map and reduce functions*

Now that we have used `map` and `reduce`, let us try and implement them ourselves. However, we want to go with an eager execution style, i.e. the return value of our `map` function should be a list instead of an iterator that is only evaluated once we iterate over it.

a) Implement your own `map` function. Its signature should be something like:

```
def map(func:  Callable[tuple,R], *iterable:Sequence) -> List[R]:
```

where `R` is a return value type variable and we do not specify, which input types the Sequences can have. We here use an interesting notation, where we do not care about how many extra arguments are provided as iterable, indicated by the star-notation. We can pass these additional arguments to a function, via e.g. `zip(*iterable)`. You can use a list comprehension to preform the necessary operations or use a for loop, whichever you prefer.
Indeed, the python type system does not allow us to easily specify that `func` needs to accept exactly as many arguments as there are sequences in `iterable`.
Validate that your implementation of `map` is correct by using it for your previous code by moving its definition to the top of your code file using map for the initial calculations.
Add an appropriate documentation comment to the function for other people to understand how it should be used (Use the NumPy/SciPy style as discussed in the lecture and laid out here: `https://numpydoc.readthedocs.io/en/latest/format.html`). Think careful about which of the sections you really need to include. E.g., if there are not exceptions thrown, you will not need a section on exceptions. However, your function will certainly require a summary, an extended description, an explanation of the parameters and the return type. Explicitly highlight the requirement for the same number of provided iterables as well as expected arguments to the `func` function! Also point out the type requirements for the sequences to match the types of the arguments for the `func` function, which we cannot easily express in terms of the typing-package's type annotations.

b) Implement your own `reduce` function. Its signature should be something like:

```
def reduce(func:  Callable[[C,N],C], iterable:Sequence[N],
                  initial:Optional[C]=None) -> C:
```

Here, we have more explicit type variables than for `map`. `C` is the return type of the reduce call and also the type of the cumulative variable in the reduce operation. `func` therefore needs to accept it as its first argument, but also return it as its result. The input sequence must be of the input type `N` that is provided to `func` as the second parameter and the initial value must be of the cumulative type.
For the implementation of `reduce`, you can either use a `for`-loop or a recursive function call. Both have their upsides and downsides, but for now, either will do.
Use your self-defined `reduce`-function in the code written for the previous problem and verify that it works as intended.
Add an appropriate documentation comment to the function for other people to understand how it should be used again in the NumPy/SciPy style. Illustrate it with an example involving the summation of all entries in an array.