Contents

Explore the tools

- Make sure you can start the various tools, directly or using the Anaconda Navigator
- Get the notebooks for the course, and play around with them (run/edit code, edit markdown cells etc.)
- Use Spyder to create a small script with at least own function
- Test (run) it, try to use the debugger, see how you can
- See how the objects and names created can now be used in the Ipython console
- Try to run the script from a console / command prompt, using both python and ipython
- To reuse the definitions in your script, you can import it into your current session (trewat it as a module) or you can start python or ipython with that script and the -i option: python -i yourscript.py. Check this out.

Modules and packages

Some of the traps you may walk into using imports and setting up packages can be found here:

http://python-notes.curiousefficiency.org/en/latest/python_concepts/import_traps.html

If working with virtual environments and/or creating distributions has no secrets for you, you may want to take a look at these.

1. Create a virtual environment, activate and install some package using pip.
2. Create trivial script using package.
3. Create requirements listing.
4. Probably for a later time: make your own distribution, put it on the test PyPI server, install it and remove it again.

See https://hynek.me/articles/sharing-your-labor-of-love-pypi-quick-and-dirty/ for general instructions. See https://packaging.python.org/guides/using-testpypi/ for instructions on how to use the test server

Mutable data

The code below is an attempt at writing a function **add** that adds a value to a list of values stored under the k key in table. When the key is new, the function should give the caller the option to specify an initial list of values to use.

There are several things that go wrong here. Why? How would you resolve this?

```
table = {}
last_added = ()
def add(k, v, start=[]):
    last_added = (k,v)
    print("Adding: ", last_added)
    if k in table:
        table[k].append(v)
    else:
        table[k] = start.append(v)
add("x", 2)
add("y", 5)
add("x", 5)
add("z",3,[2])
add("z",4)
print("Last added to table {}: {}", table, last_added)
```

Build-your-own Python Exercises

One can learn a lot from simply trying to implement some the built-in functions, such as e.g. map, filter and reduce. With theknowledge on iterators in mind, you could also try to write your own for statement (as a function).

Comprehension Exercises

1: List Comprehension

```
a = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Write a list comprehension that takes this list and makes a new list with only the even elements in this list.

2: Another List Comprehension

Find all x where x is a natural number less than or equal to 100 and x is a perfect square. This can be solved using a for-loop as:

```
for i in range(1,101):     #the iterator
   if int(i**0.5)==i**0.5: #conditional filtering
```

```
     print i               #output-expression
```

Create a list comprehension doing the same

3: Matrix Flattening

Take a 2-D matrix as input and return a list with each row placed on after the other.

The Python code with a FOR-loop could look as follows:

```
def flatten(matrix):
    flat = []
    for row in matrix:
        for x in row:
            flat.append(x)
    return flat
```

Create a matrix (using list comprehension!) and test flatten. Create a list comprehension version that achieves the same result.

Other options to pursue:

- Extend to n-dimensional matrices
- Or to trees

4: Dictionary Comprehension

Take two lists of the same length as input and return a dictionary with the first list as keys and the second as values.

The Python code with a FOR-loop could look like this:

```
def makedict(keys, values):
    dic = {}
    for i in range(len(keys)):
        dic[keys[i]] = values[i]
    return dic
```

Create a solution using dictionary comprehension, using e.g.:

country = ['Germany', 'France', 'Belgium', 'England', 'Spain', 'Italy']

capital = ['Berlin', 'Paris', 'Brussels', 'London', 'Madrid', 'Rome']

Hint: look at the `zip` generator.

Dictionary comprehension is just one way to achieve this. Look up the dict type and see what other (built-in) options there are.

Generator Exercises

1. *infinite number generator*
   Write function that generates infinite number of integers.

2. *firstn*
   Write a generator that generates first n items of an iterator.

3. *Fibonacci*
   Write Fibonacci function as generator

4. *Eratosthenes sieve* (pretty hard)
   A method for `filtering out' prime numbers, invented by the Alexandrian mathematician Eratosthenes, works as follows. Suppose you want to find out all prime numbers below, say, 1000. You first cancel all multiples of 2 (except 2) from a list 1..1000. Now you will cancel all multiples of 3 (except 3). 4 has already been canceled, as it is a multiple of 2. Now you will take off all multiples of 5, except 5. And so on. Ultimately, what remains in the list would be prime numbers!
   Implement a sieve using generators.

Decorator Exercises

Write a decorator to count the number of times a decorated function is called. Decorator should be usable on any function.

1. Write a decorator "memoize" that caches results of function calls and uses this cache to fetch pre-computed results whenever possible. Try it on e.g. a recursive Fibonacci function such as this one:

   ```
   def fib(n):
          return n if n in [0, 1] else fib(n - 2) + fib(n - 1)
   ```

   See `functools.lru_cache`(*maxsize=128*, *typed=False*) for a built-in version

2. Write a decorator that adds exception logging to a function. It only has to deal with exceptions not caught by the function.
   Extend the decorator s that it accepts a logger as argument and uses that particular logger.