



University of
Zurich^{UZH}

Faculty of Science



Scientific Programming with Python

Advanced Python Concepts Exercises

13 July 2026

Exercise 1: Playing with examples [basic]

The file `examples.zip` contains the code used in the lecture slides. Play with any example that still confuses you, until you understand how the code works.

Exercise 2: `else` on Loops and Exceptions [basic]

The file `else_loop.py` contains a list with random integers. A typical use-case for `else` on loops is a search algorithm, that will need to handle the case where the search-target is not found in the searched objects.

- Implement a search function that takes a list, array or tuple of integers and looks for a sequence of at least N identical, consecutive values. If no sequence is found, the function should return a `LookupError`. Otherwise it should return the starting index and the length of the sequence.
- Create an alternative version of the function from (a) that does/does not use `else` on loops. (Depending on how you solved (a).) Compare advantages and disadvantages.
- Use the function from (a) to find and print all sequences of at least length 3 in the input data.

Exercise 3: Processing Measurements with `match` [basic]

- Implement a function `to_celsius`. The function should take two arguments
 - the value of the measurement
 - the current unit ("`K`", "`°F`" or "`°R`")and tuple with the value converted to degree Celsius and "`°C`"
- The file `measurements.py` contains a stream of readouts from a climate sensor. Each readout contains measurements for temperature in Kelvin, relative humidity as a fraction and air pressure in Pascal.

Use `match` to write a function that takes a readout, loops over the measurements and convert each value into a more user-friendly form

temperature to degrees Celsius

humidity to percent

pressure to hPa

- (c) Solve the task from (b) using `if/elif` as well as a dictionary mapping the measurement keys to conversion functions. Compare advantages and disadvantages.

Exercise 4: matching Times [intermediate]

The file `matching_time.py` generates a list of hourly timestamps. Process these timestamps and print a message depending on the day of the month and the time of the day. As examples, you could print

- a message at the first of each month
- **weekend** if it's a Saturday or Sunday
- **lunch time** if the hour is 12
- **working** if the hour is between 9 and 17
- **relaxing** at any other time

Exercise 5: Exception Basics [basic]

- (a) Write code that reads input from the user and tries to convert it first into an `int` and then into a `float`. If both fail raise an `InputError`. (You need to create that class yourself, choose an appropriate base error.)
- (b) Extend (a) to also try `complex` and `fractions.Fraction`
- (c) Take the complex example from the slides (in `exceptions_example.py`) and reorganise the code, such that the different problems are raised and handled in separate `try ...except` blocks.

Exercise 6: Raising Exceptions [intermediate]

- (a) The file `raising_exceptions.py` contains a long string of lowercase characters. Write a function `assert_ascii` that checks the input. If it contains characters not in `string.ascii_lowercase`, raise an appropriate error and pass back the position of the offending character as a second argument.
- (b) Use `assert_ascii` from (a) and a loop to remove all illegal characters from the string.
- (c) Extend (b) and count the number of illegal characters removed. If too many characters get removed, raise a new error.

- (d) Compare the effect of the different options to chain errors by adjusting the way you raise the new error (i.e. with and without `from ...`). Also check the effect of `real bug` in the exception code.

Exercise 7: Timing Function [intermediate]

Create a timing function `fime_func` that works as follows

```
>>> timed_pow = fime_func(pow)
>>> timed_pow(2, 42)
time needed: ...
4398046511104
```

- Assure your self, that all arguments and keyword arguments of the original function still work.
- Try using `fime_func` as a decorator (i.e. define a custom function and apply `@fime_func` to the definition)

Exercise 8: Decorator with Argument [advanced]

Write a decorator that takes a message as argument. When the decorated function is called, first the message should be printed before the decorated function is executed.

Hint: `@myfunction()` first evaluates `myfunction()`, then uses the result as the decorator.

Exercise 9: Anonymised Context [intermediate]

Given dictionaries like `{"name": "john", "age": 23, "height": 171}` Create a context manager that *temporarily* removes the "name" item and puts it back on exit.

Exercise 10: Collatz sequence [intermediate]

Create an iterator that, given any positive integer as starting value, returns the numbers from Collatz conjecture. Implement your solution first using an iterator-class and then a generator function.