Deep Learning — Lab 1-3

Introduction to Python

T. Méndez / H. Badertscher

HS 2022/23

1 Topic

In the first two lab sessions we will introduce the basics of Python and some Python libraries (NumPy and Matplotlib), as many deep learning libraries (e.g. TensorFlow, Theano, etc.) are python libraries. For this introduction we basically follow the Python tutorial from `https://docs.python.org/3/tutorial/index.html` and the numpy tutorial from `http://cs231n.github.io/python-numpy-tutorial/` respectively. Some parts of the tutorial that are not important in the context of deep learning are omitted. Instead, the tutorial has been supplemented with some exercises, which should be worked on for more in-depth study. Also try out the commands, as this is a good way to learn a new language.

There are currently two different supported versions of Python, 2.7 and 3.5. Somewhat confusingly, Python 3.0 introduced many backwards-incompatible changes to the language, so code written for 2.7 may not work under 3.5 and vice versa. For this class all code will use Python 3.5.

2 Spyder

We recommend to work with the Python development environment *Spyder*. This is a very lean development environment that is well suited for beginners. An overview of Spyder and its main features is shown in Figure 1.

3 Python Tutorial

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

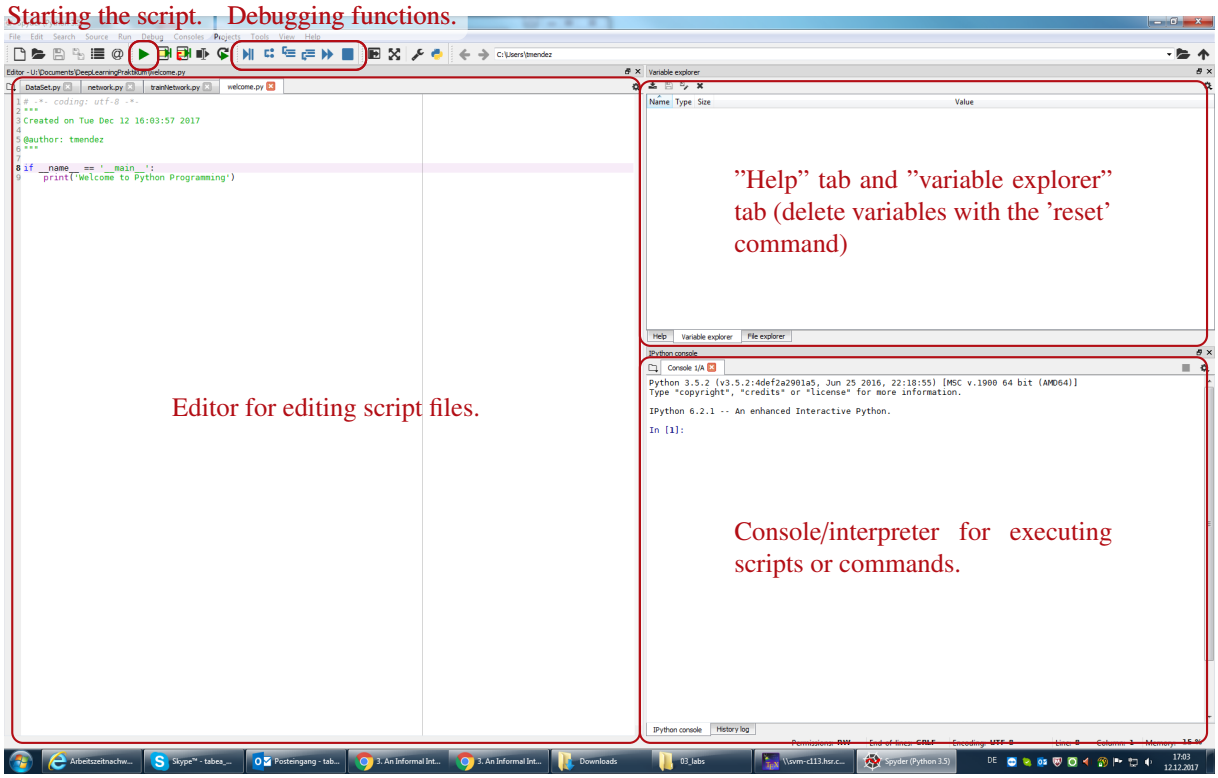

Figure 1: Overview of the development environment Spyder and its main features.

This tutorial introduces the reader informally to the basic concepts and features of the Python language and system. It does not attempt to be comprehensive and cover every single feature, or even every commonly used feature. Instead, it introduces many of Python’s most noteworthy features, and will give you a good idea of the language’s flavor and style. After reading it, you will be able to read and write Python modules and programs, and you will be ready to learn more about the various Python library modules described in The Python Standard Library.

3.1 An Informal Introduction to Python

In the following examples, input and output are distinguished by the presence or absence of prompts (>>> and ...): to repeat the example, you must type everything after the prompt, when the prompt appears; lines that do not begin with a prompt are output from the interpreter. Note that a secondary prompt on a line by itself in an example means you must type a blank line; this is used to end a multi-line command.

Many of the examples in this manual, even those entered at the interactive prompt, include comments. Comments in Python start with the hash character, #, and extend to the end of the physical line. A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character. Since comments are to clarify code and are not interpreted by Python, they may be omitted when typing in examples.

Some examples:

```
# this is the first comment
spam = 1  # and this is the second comment
          # ... and now a third!
text = "# This is not a comment because it's inside quotes."
```

Let's try some simple Python commands. Start Spyder or an other interpreter and wait for the primary prompt, >>>. When using Spyder the prompt corresponds to the string `In [1]:` in the console.

3.1.1 Numbers

The interpreter acts as a simple calculator. You can type an expression at it and it will write the value. Expression syntax is straightforward, for example.

- The operators `+`, `-`, `*` and `/` work just like in most other languages (for example C); parentheses `()` can be used for grouping.

```
>>> 50 + 5*6
80
>>> (50 - 5*6) / 4
5.0
```

- The integer numbers (e.g. `2`, `4`, `20`) have type `int`, the ones with a fractional part (e.g. `5.0`, `1.6`) have type `float`.

- Division (`/`) always returns a float. To do floor division and get an integer result (discarding any fractional result) you can use the `//` operator; to calculate the remainder you can use `%`.

```
>>> 17 / 3     # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3     # floor division discards the fractional part
5
>>> 17 % 3      # the % operator returns the remainder of the division
2
```

- With Python, it is possible to use the ** operator to calculate powers.

```
>>> 5 ** 2      # 5 squared
25
>>> 2 ** 7      # 2 to the power of 7
128
```

- The equal sign (=) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt.

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

3.1.2 Strings

Besides numbers, Python can also manipulate strings, which can be expressed in several ways.

- Strings can be enclosed in single quotes (`'...'`) or double quotes (`"..."`) with the same result. \ can be used to escape quotes.

```
>>> 'spam eggs'    # single quotes
'spam eggs'
>>> 'doesn\'t'     # use \' to escape the single quote...
"doesn't"
>>> "doesn't"      # ...or use double quotes instead
"doesn't"
```

- In the interactive interpreter, the output string is enclosed in quotes and special characters are escaped with backslashes. The `print()` function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters.

```
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
>>> print('"Isn\'t," she said.')
"Isn't," she said.
>>> s = 'First line.\nSecond line.' # \n means newline
>>> s                          # without print(), \n is included in the output
'First line.\nSecond line.'
>>> print(s)                    # with print(), \n produces a new line
First line.
Second line.
```

- Strings can be concatenated (glued together) with the + operator, and repeated with *.

```
>>> prefix = 'un'
>>> 3 * prefix + 'ium'     # 3 times 'un', followed by 'ium'
'unununium'
```

- Two or more string literals next to each other are automatically concatenated. This feature is particularly useful when you want to break long strings.

```
>>> text = ('Put several strings within parentheses '
...         'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

- Strings can be indexed (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one.

```
>>> word = 'Python'
>>> word[0]     # character in position 0
'P'
>>> word[5]     # character in position 5
'n'
```

- Indices may also be negative numbers, to start counting from the right. Note that since -0 is the same as 0, negative indices start from -1.

```
>>> word[-1]  # last character
'n'
>>> word[-2]  # second-last character
'o'
>>> word[-6]
'P'
```

- In addition to indexing, slicing is also supported. While indexing is used to obtain individual characters, slicing allows you to obtain substring.

```
>>> word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5]  # characters from position 2 (included) to 5 (excluded)
'tho'
```

- Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

```
>>> word[:2]   # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:]   # characters from position 4 (included) to the end
'on'
>>> word[-2:]  # characters from the second-last (included) to the end
'on'
```

- One way to remember how slices work is to think of the indices as pointing between characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of n characters has index n, for example.

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

- Python strings cannot be changed — they are immutable. Therefore, assigning to an indexed position in the string results in an error. If you need a different string, you should create a new one.

```
>>> word[0] = 'J'
...
TypeError: 'str' object does not support item assignment
>>> 'J' + word[1:]
'Jython'
```

- The built-in function `len()` returns the length of a string.

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

- With the built-in function `str()` numbers can be converted into strings.

```
>>> n = 37.5
>>> print('n is equal to ' + str(n))
n is equal to 37.5
```

- With the built-in functions `int()` and `float()` strings can be converted into numbers.

```
>>> var1 = int('375')
>>> var1
357
>>> var2 = float('375.75')
>>> var2
375.75
```

- To find a substring in a string, the method `find()` can be used.

```
>>> s = 'Hello Word'
>>> s.find('ll')
2
```

3.1.3 Lists

Python knows a number of compound data types, used to group together other values. The most versatile is the list, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type.

- Like strings (and all other built-in sequence type), lists can be indexed and sliced. All slice operations return a new list containing the requested elements.

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
>>> squares[0]      # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]    # slicing returns a new list
[9, 16, 25]
```

- Lists also support operations like concatenation.

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- Unlike strings, which are immutable, lists are a mutable type, i.e. it is possible to change their content.

```
>>> cubes = [1, 8, 27, 65, 125]     # something's wrong here
>>> 4 ** 3                         # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 4 ** 3              # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

- You can also add new items at the end of the list, by using the `append()` method.

```
>>> cubes.append(216)       # add the cube of 6
>>> cubes.append(7 ** 3)    # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

- Assignment to slices is also possible, and this can even change the size of the list or clear it entirely.

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

- The built-in function `len()` also applies to lists.

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

- It is possible to nest lists (create lists containing other lists), for example:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

3.2 First Steps Towards Programming

Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write a sequence of operations that are logically linked to each other using control flow tools such as while, if, or for. When writing such sequences or an entire script, it is more convenient to work with the editor than writing code directly in the console/interpreter. When working with Spyder, code can be written in the editor (left) and executed in the interpreter (right) by pressing the green play-icon. As an example, we can write an initial mini script which calculates and prints the Fibonacci as follows:

```
1 if __name__ == "__main__":
2     # Fibonacci series:
```

```
3      # the sum of two elements defines the next
4      print('Fibonacci series:')
5      a, b = 0, 1
6      while b < 10:
7          print('nr: ', b)
8          a, b = b, a+b
```

If this script is executed, the following output results:

```
Fibonacci series:
nr: 1
nr: 1
nr: 2
nr: 3
nr: 5
nr: 8
```

This example introduces several new features of Python:

- When a source file is read by the Python interpreter, all the code it contains is executed. Unlike other languages, Python does not have a `main()` function that runs automatically. However, some special variables are defined before the code is executed. The special variable `__name__` is set to the value `"__main__"` if the Python interpreter executes this module (source file) as the main program. However, if the file is imported from another module, the variable `__name__` is set to the name of the module. If you check this variable, you can find out whether the file is running as the main program or not. The block within the `if` statement is only executed if the source file was called as the main program.
- The body of the `if` statement and the while-loop is indented: **Indentation is Python's way of grouping statements.** Unlike in C, where the grouping of blocks is denoted by the brackets { and }, in Python the grouping of blocks is only denoted by the indentation. Note that each line within a basic block must be indented by the same amount. It is recommended and common practice to use 4-space indentation and not tabs. 4 spaces are a good compromise between small indentation (allows greater nesting depth) and large indentation (easier to read). Tabs lead to confusion and should be omitted.
- The 5th line contains a multiple assignment: the variables a and b simultaneously get the new values 0 and 1. On the last line this is used again, demonstrating that the expressions on the right-hand side are all evaluated first before any of the assignments take place. The right-hand side expressions are evaluated from the left to the right.
- The while loop executes as long as the condition (here: `b < 10`) remains true. In Python, like in C, any non-zero integer value is true; zero is false. The condition may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false. The test used in the example is a simple comparison. The standard comparison operators are written the same as in C:
  - less than: <
  - greater than: >
  - equal to: ==

  - less than or equal to: <=
  - greater than or equal to: >=
  - not equal to: !=

  Python also provides some additional operators:

  - Returns `True` if all elements of the *iterable* (e.g. a list) are true: `all(<iterable>)`
  - Returns `True` if any element of the *iterable* (e.g. a list) is true: `any(<iterable>)`

- The `print()` function writes the value of the argument(s) it is given. It differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles multiple arguments, floating point quantities, and strings. Strings are printed without quotes, and a space is inserted between items, so you can format things nicely.

  The keyword argument `end` can be used to avoid the newline after the output, or end the output with a different string:

```
a, b = 0, 1
while b < 1000:
    print(b, end=',')
    a, b = b, a+b
```

  generates the output:

```
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

3.3 Control Flow Tools

Besides the `while` statement just introduced, Python knows the usual control flow statements known from other languages, with some twists.

3.3.1 `if` Statements

Perhaps the most well-known statement type is the `if` statement. An example is given below:

```
if __name__ == "__main__":
    # an example of an if statement
    x = int(input("Please enter an integer: "))
    if x < 0:
        x = 0
        print('Negative changed to zero')
    elif x == 0:
        print('Zero')
    elif x == 1:
        print('Single')
    else:
        print('More')
```

It generates the following output:

```
Please enter an integer: 42
More
```

There can be zero or more `elif` parts, and the `else` part is optional. The keyword `elif` is short for ”else if”, and is useful to avoid excessive indentation. An `if`...`elif`...`elif`... sequence is a substitute for the `switch-case` statement found in other languages.

3.3.2 `for` Statements

The `for` statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python’s `for` statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example:

```
1 if __name__ == "__main__":
2     # Measure some strings:
3     words = ['cat', 'window', 'defenestrate']
4     for w in words:
5         print(w, len(w))
```

The output of this script is:

```
cat 3
window 6
defenestrate 12
```

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates arithmetic progressions:

```
1 if __name__ == "__main__":
2     for i in range(5):
3         print(i)
```

Which generates the output:

```
0
1
2
3
4
```

The given end point is never part of the generated sequence; `range(10)` generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the ”step”):

- `range(5, 10)` → `5, 6, 7, 8, 9`
- `range(0, 10, 3)` → `0, 3, 6, 9`
- `range(-10, -100, -30)` → `-10, -40, -70`

To iterate over the indices of a sequence, you can combine `range()` and `len()` as follows:

```
if __name__ == "__main__":
    a = ['Mary', 'had', 'a', 'little', 'lamb']
    for i in range(len(a)):
        print(i, a[i])
```

Which generates the output:

```
0 Mary
1 had
2 a
3 little
4 lamb
```

In many ways the object returned by `range()` behaves as if it is a list, but in fact it isn't. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn't really make the list, thus saving space. We say such an object is iterable, that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted. We have seen that the for statement is such an iterator.

3.3.3 `break` and `continue` Statements

The `break` statement, like in C, breaks out of the innermost enclosing `for` or `while` loop. The `continue` statement, also borrowed from C, continues with the next iteration of the loop. An example is given below:

```
if __name__ == "__main__":
    for n in range(2, 10):
        if n % 2 == 0:
            print("Found an even number:", n)
            continue

        if n > 5:
            print("Found an odd number greater then 5:", n)
            break
```

It generates the following output:

```
Found an even number: 2
Found an even number: 4
Found an even number: 6
Found an odd number greater then 5: 7
```

3.4 Exercise 1

Write a Python script that asks the user to enter a year and check if the entered year is a leap year. In response, output a string such as "The year XXX is not a leap year."Design your program in such a way that several numbers can be entered together, as shown in the following example:

```
Enter one or more years: 1992 1995 2000 2100
The year 1992 is a leap year.
The year 1995 is not a leap year.
The year 2000 is a leap year.
The year 2100 is not a leap year.
```

When you are done, Try the following year dates to test your program:

- 2010 → no leap year
- 2008 → leap year
- 1700 → no leap year
- 1701 → no leap year
- 1702 → no leap year
- 1703 → no leap year
- 1704 → leap year
- 1600 → leap year
- 1584 → leap year
- -1584 → invalid input

Hint:
To read in the numbers, the built-in function `input()` can be used.

3.5 Defining Functions

In Python we can also define functions. Functions must always be defined first, before they can be used. For this reason, they are always placed ahead of the statement `if __name__ == "__main__":` .

As an example, we can create a function that writes the Fibonacci series to an arbitrary boundary as follows:

```
1  def fib(n):
2      """ Print a Fibonacci series up to n.
3
4          Args:
5              n:  number up to which the series is printed.
6      """
7      a, b = 0, 1
8      while a < n:
9          print(a, end=' ')
10         a, b = b, a+b
11     print()
12
13 if __name__ == "__main__":
14     # call the function we just defined:
15     fib(2000)
```

The output of this script is:

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword `def` introduces a function definition. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or docstring. It's good practice to include docstrings in code that you write, so make a habit of it.

The execution of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables cannot be directly assigned a value within a function (unless named in a `global` statement), although they may be referenced.

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using *call by value*, where the *value* is always an object *reference*, not the value of the object). Actually, *call by object reference* would be a better description, since if a mutable object is passed, the caller will see any changes the callee makes to it (items inserted into a list). When a function calls another function, a new local symbol table is created for that call.

Coming from other languages, you might object that `fib` is not a function but a procedure since it doesn't return a value. In fact, even functions without a return statement do return a value, albeit a rather boring one. This value is called `None` (it's a built-in name). Writing the value `None` is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to using print():

```
>>> print(fib(0))
None
```

The `fib` function can be easily modified to return a list of Fibonacci series numbers instead of printing them:

```
def fib2(n):
    """ Return a list containing the Fibonacci series up to n.

        Args:
            n:  number up to which the series is calculated.
        Return:
            list containing the Fibonacci series.
    """
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b

    return result

if __name__ == "__main__":
    # call the function we just defined:
    fibSer = fib2(200)
    print('Fibonacci series', fibSer)
```

The output of this script is:

```
Fibonacci series [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
```

This example, as usual, demonstrates some new Python features:

- The `return` statement returns with a value from a function. `return` without an expression argument returns `None`. Falling off the end of a function also returns `None`.
- The statement `result.append(a)` calls a *method* of the list object `result`. A method is a function that "belongs" to an object and is named `obj.methodname`, where `obj` is some object (this may be an expression), and `methodname` is the name of a method that is defined by the object's type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. The method `append()` shown in the example is defined for list objects; it adds a new element at the end of the list.

3.5.1 Argument Values

It is also possible to define functions with a variable number of arguments. There are different forms, which can be combined.

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined. For example, the following function :

```
def sentence(time, action='walking', name='Peter', place='Paris'):
    print(name, 'has been', action, 'in', place, 'for', time, 'hours.')
```

can be called in several ways:

- giving only the mandatory argument:

```
>>> sentence(3)
Peter has been walking in Paris for 3 hours.
```

- giving some of the optional arguments:

```
>>> sentence(3, 'jumping', 'Monika')
Monika has been jumping in Paris for 3 hours.
```

- giving all arguments:

```
>>> sentence(3, 'jumping', 'Monika', 'New York')
Monika has been jumping in New York for 3 hours.
```

The default values are evaluated at the point of function definition in the defining scope.

Functions can also be called using keyword arguments of the form `kwarg=value`. For instance, the previous function, which accepts one required argument (`time`) and three optional arguments (`action, name, and place`), can be called in any of the following ways:

- 1 positional argument:

```
>>> sentence(3)
Peter has been walking in Paris for 3 hours.
```

- 1 keyword argument:

```
>>> sentence(time=3)
Peter has been walking in Paris for 3 hours.
```

- 2 keyword arguments:

```
>>> sentence(time=3, action='jumping')
Peter has been jumping in Paris for 3 hours.
```

- 2 keyword arguments:

```
>>> sentence(action='jumping', time=3)
Peter has been jumping in Paris for 3 hours.
```

- 3 positional arguments:

```
>>> sentence(3, 'jumping', 'Monika')
Monika has been jumping in Paris for 3 hours.
```

- 1 positional, 1 keyword:

```
>>> sentence(3, action='jumping')
Peter has been jumping in Paris for 3 hours.
```

but all the following calls would be invalid:

- required argument missing:

```
>>> sentence()
```

- non-keyword argument after a keyword argument:

```
>>> sentence(time=3, 'jumping')
```

- duplicate value for the same argument:

```
>>> sentence(3, time=3)
```

- unknown keyword argument:

```
>>> sentence(actor='Monika')
```

In a function call, keyword arguments must follow positional arguments. All the keyword arguments passed must match one of the arguments accepted by the function (e.g. `actor` is not a valid argument for the `sentence` function), and their order is not important. This also includes non-optional arguments (e.g. `sentence(time=3)` is valid too). No argument may receive a value more than once.

3.6 Modules

So far we have only written very small scripts in the text editor, where the whole code was in one file. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you have written in several programs without copying its definition into each program. To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a module; definitions from a module can be imported into other modules or into the main module.

Thus, a module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. For instance, use Sypder to create a module (file) called `fibo.py` in your lab directory (e.g. named `IntroductionToPython`) with the following content:

```
# Fibonacci numbers module

def fib(n):
    """ Print a Fibonacci series up to n.

        Args:
            n:  number up to which the series is printed.
    """
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):
    """ Return a list containing the Fibonacci series up to n.

        Args:
            n:  number up to which the series is calculated.
        Return:
            list containing the Fibonacci series.
    """
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b

    return result


if __name__ == "__main__":
    # testing the functions of the module fibo
    fib(1000)
    fibSer = fib2(100)
    print('Fibonacci series', fibSer)
```

This module `fibo` can either be used as a script by running it directly as the main file, or it can be imported into other modules or into the main module.

When this module is executed as the main file, the global variable `__name__` is set to the value `"__main__"`, whereby the block within the `if` statement is executed. However, if this module is imported by another module, the global variable `__name__` is set to module's name `"fibo"`, so that the block within the `if` statement is NOT executed. In this way, you can use the file both as a script and as an importable module. This is often used to provide a convenient user interface for a module, or to use it for testing purposes (running the module as a script executes a test suite).

The output, when executed as the main file, is:

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
Fibonacci series [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

where the first line is generated by the function `fib` and the second line by the `print` command.

Now create an other file called `master.py` in the same directory that imports the module `fibo` as follows:

```
1  import fibo
2
3  def main():
4      # call the functions form the module fibo
5      fibo.fib(2000)
6      fibSer = fibo.fib2(200)
7      print('Fibonacci series', fibSer)
8
9  if __name__ == "__main__":
10     main()
```

This import statement does not enter the names of the functions defined in `fibo` directly in the current symbol table; it only enters the module name `fibo` there. Using the module name you can access the functions as shown above.

The output generated when the `master` script is executed is:

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
Fibonacci series [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
```

Note that the output is different form the one above, since the arguments of the functions are different.

Note also that in the `if` statement only the `main()` function is called. This way, the variables of the `main()` function are better encapsulated, since variables within a function are local, while variables outside a function are global. This makes the code cleaner and better organized. It also allows you to use the `main()` function in other modules.

3.6.1 More on Modules

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the **first** time the module name is encountered in an `import` statement.

Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables.

Modules can import other modules. It is common practice to place all `import` statements at the beginning of a module (or script, for that matter). The imported module names are placed in the importing module's global symbol table.

There is a variant of the `import` statement that imports names from a module directly into the importing module's symbol table. For example:

```
from fibo import fib, fib2

def main():
    # call the functions form the module fibo
    fib(1000)
    fibSer = fib2(100)
    print('Fibonacci series', fibSer)

if __name__ == "__main__":
    main()
```

This does not introduce the module name from which the imports are taken in the local symbol table. So in the example, `fibo` is not defined.

To save typing work, the module can also be renamed during import. For example:

```
import fibo as fi

def main():
    # call the functions form the module fibo
    fi.fib(2000)
    fibSer = fi.fib2(200)
    print('Fibonacci series', fibSer)

if __name__ == "__main__":
    main()
```

As a result, the old module name (`fibo`) is no longer available.

3.6.2 Module Search Path

When a module (e.g. named `spam`) is imported, the interpreter first searches for a built-in module with that name. If not found, it then searches for a file named `spam.py` in a list of directories given by the variable `sys.path`. The variable `sys.path` is initialized with the following paths by default:

- The directory containing the input script
- The standard library path

After initialization, Python programs can modify `sys.path`. The directory containing the script being run is placed at the beginning of the search path, ahead of the standard library path. This means that scripts in that directory will be loaded instead of modules of the same name in the library directory. This is an error unless the replacement is intended.

3.6.3 Standard Modules

Python comes with a library of standard modules. Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the underlying platform. For example, the `winreg` module is only provided on Windows systems. Often used standard modules are:

- `sys:` Access system-specific parameters and functions
- `math:` Mathematical functions (sin() etc.).
- `random:` Generate pseudo-random numbers with various common distributions.

3.6.4 Packages

Packages are a way of structuring Python's module namespace by using "dotted module names". For example, the module name `A.B` designates a submodule named `B` in a package named `A`. Just like the use of modules saves the authors of different modules from having to worry about each other's global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or the Python Imaging Library from having to worry about each other's module names.

Suppose you want to design a collection of modules (a package) for the uniform handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, for example: `.wav`, `.aiff`, `.au`), so you may need to create and maintain a growing collection of modules for the conversion between the various file formats. There are also many different operations you might want to perform on sound data, such as mixing, adding echo, and so on. So in addition you will be writing a never-ending stream of modules to perform these operations. Here is a possible structure for your package, expressed in terms of a hierarchical file system:

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
```

```
            ...
    effects/                Subpackage for sound effects
            __init__.py
            echo.py
            surround.py
            reverse.py
            ...
    filters/                Subpackage for filters
            __init__.py
            equalizer.py
            vocoder.py
            karaoke.py
            ...
```

When importing this package, Python searches through the directories on `sys.path` looking for the package subdirectory. The `__init__.py` files are required to make Python treat the directories as containing packages; this is done to prevent directories with a common name, such as `string`, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package.

Users of the package can import individual modules from the package, for example:

```
import sound.effects.echo
```

This loads the submodule sound.effects.echo. If a function of this submodule is used, it must be referenced with its full name.

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

To avoid this, the submodule can alternatively be imported as follows:

```
from sound.effects import echo
```

This also loads the submodule `echo`, and makes it available without its package prefix, so it can be used as follows:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Note that when using `from package import item`, the `item` can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable. The `import` statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an `ImportError` exception is raised.

Contrarily, when using syntax like `import item.subitem.subsubitem`, each item except for the last must be a package; the last item can be a module or a package but can't be a class or function or variable defined in the previous item.

3.7 Output

There are several ways to present the output of a program; data can be printed in a human-readable form, or written to a file for future use. In this chapter, we will discuss in more detail the possibilities of printing formatted strings using the `str.format()` method.

Basic usage of the `str.format()` method looks like this:

```
>>> print('We are the {} who say "{}!"'.format('knights', 'Ni'))
We are the knights who say "Ni!"
```

The brackets and characters within them (called format fields) are replaced with the objects passed into the `str.format()` method. A number in the brackets can be used to refer to the position of the object passed into the `str.format()` method.

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

If keyword arguments are used in the `str.format()` method, their values are referred to by using the name of the argument.

```
>>> print('This {food} is {adjective}.'.format(
...       food='spam', adjective='absolutely horrible'))
This spam is absolutely horrible.
```

An optional `':'` and format specifier can follow the field name. This allows greater control over how the value is formatted. The following example rounds Pi to three places after the decimal.

```
>>> import math
>>> print('The value of PI is approximately {0:.3f}.'.format(math.pi))
The value of PI is approximately 3.142.
```

Passing an integer after the `':'` will cause that field to be a minimum number of characters wide.

```
>>> names = ['Sjoerd', 'Jack', 'Dcab']
>>> age = [34, 87, 5]
>>> size = [1.874, 1.752, 1.153]
>>> for i in range(len(age)):
...     print('Name: {0:7}  /  Alter: {1:4d}  /  Groesse: {2:6.2f}'.format(
...           names[i], age[i], size[i]))
...
Name: Sjoerd   /  Alter:   34  /  Groesse:   1.87
Name: Jack     /  Alter:   87  /  Groesse:   1.75
Name: Dcab     /  Alter:    5  /  Groesse:   1.15
```

3.8 Exercise 2

Modify your script from last week (determining whether a year is a leap year) so that it can be used as a function. The function should accept a list of years as an argument and return

an equally long list of boolean values. The boolean value at the n-th position in the return list belongs to the year at the n-th position in the given input list and indicates whether the corresponding year is a leap year. The default value of the argument of the function should be a list with only one year, the current one. Create your own module called `leapYear.py` for this function.

To test the module `leapYear.py`, write a test-suite (a "main-if-block" in the module file), which checks the following cases:

- Calling the function with the default argument.
- Calling the function with only one element in the list. Test all years given in exercise 1.
- Calling the function with a list containing all the years given in Exercise 1.

Now create a new main file (e. g. `master.py`), which imports the module `leapYear.py` and asks the user to enter a year. The user interface should be identical with the one in exercise 1. To check whether the entered years are leap years, the function of the module `master.py` should now be used.

3.9 The NumPy Library

NumPy is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.

3.9.1 Arrays

A NumPy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. In NumPy dimensions are called axes. The number of axes is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension/axis.

For example, the coordinates of a point in 3D space [1, 2, 1] is an array of rank 1, because it has one axis. That axis has a length of 3. In the example pictured below, the array has rank 2 (it is 2-dimensional). The first dimension (axis) has a length of 2, the second dimension has a length of 3.

```
[[ 1., 0., 0.],
[ 0., 1., 2.]]
```

There are several ways to create a numpy array:

```
import numpy as np

# Create a rank 1 array
a = np.array([1, 2, 3])

# Create a rank 2 array
b = np.array([[1,2,3],[4,5,6]])
```

```
# Create a 2x2 array of all zeros
c = np.zeros((2,2))

# Create a 1x2 array of all ones
d = np.ones((1,2))

# Create a constant vector of length 2 containing only sevens
e = np.full((2,), 7)

# Create a 2x2 identity matrix
f = np.eye(2)

# Create a 2x2 array filled with random values
g = np.random.random((2,2))
```

To obtain the rank or shape of an array, the following attributes can be used:

```
>>> print(a)
[1, 2, 3]
>>> print(a.shape)
(3,)
>>> print(a.ndim)
1
>>> print(b)
[[1 2 3]
[4 5 6]]
>>> print(b.shape)
(2, 3)
>>> print(b.ndim)
2
```

Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])   # Prints "2"
b[0, 0] = 77     # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])   # Prints "77"
```

3.9.2 Array math

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

```
import numpy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

# Elementwise sum
print(x + y)
print(np.add(x, y))

# Elementwise difference
print(x - y)
print(np.subtract(x, y))

# Elementwise product
print(x * y)
print(np.multiply(x, y))

# Elementwise division
print(x / y)
print(np.divide(x, y))

# Elementwise exponentiate
print(x ** 4)
print(np.power(x, 4))

# Elementwise square root
print(np.sqrt(x))

# Elementwise sine and cosine
print(np.sin(x))
print(np.cos(x))

# Elementwise exponential function
print(np.exp(x))
```

Note that unlike MATLAB, * is elementwise multiplication, not matrix multiplication. We instead use the `matmul` function to compute the matrix product.

```
import numpy as np

A = np.array([[1,2],[3,4]])
B = np.array([[5,6],[7,8]])
x = np.array([9,10])

# Matrix Multiplication
C = np.matmul(A,B)

# Multiplication with a column vector from the right
y1 = np.matmul(A,x)

# Multiplication with a row vector from the left
y2 = np.matmul(x,A)
```

Numpy provides many useful functions for performing computations on arrays; some of the most useful ones are the `sum` and the `mean`:

```
import numpy as np

x = np.array([[1,2],[3,4]])

print(np.sum(x))            # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))    # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))    # Compute sum of each row; prints "[3 7]"

print(np.mean(x))           # Compute mean of all elements; prints "2.5"
print(np.mean(x, axis=0))   # Compute mean of each column; prints "[2.  3.]"
print(np.mean(x, axis=1))   # Compute mean of each row; prints "[1.5  3.5]"
```

You can find the full list of mathematical functions provided by numpy in the documentation (`https://docs.scipy.org/doc/numpy/reference/routines.math.html`).

Apart from computing mathematical functions using arrays, we frequently need to reshape or otherwise manipulate data in arrays. The simplest example of this type of operation is transposing a matrix; to transpose a matrix, simply use the `T` attribute of an array object:

```
import numpy as np

x = np.array([[1,2], [3,4]])
print(x)    # Prints "[[1 2]
            #          [3 4]]"
print(x.T)  # Prints "[[1 3]
            #          [2 4]]"
```

Numpy provides many more functions for manipulating arrays; you can see the full list in the documentation (`https://docs.scipy.org/doc/numpy/reference/routines.array-manipulation.html`).

3.10 Broadcasting

Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

For example, suppose that we want to add a constant vector to each row of a matrix. We could do it like this:

```
import numpy as np

# We will add the vector v to each row of the matrix X,
# storing the result in the matrix Y
X = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
Y = np.empty_like(X)   # Create an empty matrix with the same shape as X

# Add the vector v to each row of the matrix X with an explicit loop
```

```
for i in range(4):
    Y[i, :] = X[i, :] + v

# Now Y is the following
# [[ 2  2  4]
#  [ 5  5  7]
#  [ 8  8 10]
#  [11 11 13]]
print(Y)
```

This works; however when the matrix `X` is very large, computing an explicit loop in Python could be slow. Note that adding the vector `v` to each row of the matrix `X` is equivalent to forming a matrix `vv` by stacking multiple copies of `v` vertically, then performing elementwise summation of `X` and `vv`. We could implement this approach like this:

```
import numpy as np

# We will add the vector v to each row of the matrix Xx,
# storing the result in the matrix Y
X = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
vv = np.tile(v, (4, 1))   # Stack 4 copies of v on top of each other
print(vv)                 # Prints "[[1 0 1]
                          #          [1 0 1]
                          #          [1 0 1]
                          #          [1 0 1]]"
Y = X + vv  # Add x and vv elementwise
print(Y)  # Prints "[[ 2  2  4
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"
```

Numpy broadcasting allows us to perform this computation without actually creating multiple copies of `v`. Consider this version, using broadcasting:

```
import numpy as np

# We will add the vector v to each row of the matrix X,
# storing the result in the matrix Y
X = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
Y = X + v  # Add v to each row of X using broadcasting
print(Y)  # Prints "[[ 2  2  4]
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"
```

The line `Y = X + v` works even though `X` has shape `(4, 3)` and `v` has shape `(3,)` due to broadcasting; this line works as if `v` actually had shape `(4, 3)`, where each row was a copy of `v`, and the sum was performed elementwise.

3.11 Exercise 3

Use the Numpy library to perform the following calculations.

- Create the following vectors and matrices (camera model matrices):

$$\boldsymbol{K} = \begin{pmatrix} 100 & 0 & 60 \\ 0 & 100 & 45 \\ 0 & 0 & 1 \end{pmatrix}, \quad \boldsymbol{D}_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \boldsymbol{c}_1 = \begin{pmatrix} 2 \\ 2 \\ 2 \end{pmatrix}, \quad \boldsymbol{b}_1 = \begin{pmatrix} 113 \\ 10 \\ 1 \end{pmatrix},$$

$$\boldsymbol{D}_2 = \begin{pmatrix} \cos(-\pi/3) & 0 & \sin(-\pi/3) \\ 0 & 1 & 0 \\ -\sin(-\pi/3) & 0 & \cos(-\pi/3) \end{pmatrix}, \quad \boldsymbol{c}_2 = \begin{pmatrix} -100 \\ -100 \\ 100 \end{pmatrix}, \quad \boldsymbol{b}_2 = \begin{pmatrix} 86 \\ 73 \\ 1 \end{pmatrix}.$$

- Calculate the two directional vectors

$$\boldsymbol{r}_1 = (\boldsymbol{K}\boldsymbol{D}_1)^{-1}\boldsymbol{b}_1 \quad \text{and} \quad \boldsymbol{r}_2 = (\boldsymbol{K}\boldsymbol{D}_2)^{-1}\boldsymbol{b}_2.$$

  The function `np.linalg.inv` can be used for matrix inversion.

- Create the left and right sides of the equation system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{c}$ with

$$\boldsymbol{A} = \begin{pmatrix} \boldsymbol{E} & -\boldsymbol{r}_1 & 0 \\ \boldsymbol{E} & 0 & -\boldsymbol{r}_2 \end{pmatrix} \quad \text{and} \quad \boldsymbol{c} = \begin{pmatrix} \boldsymbol{c}_1 \\ \boldsymbol{c}_2 \end{pmatrix}$$

- Find the least-square solution of the equation system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{c}$ as

$$\boldsymbol{x} = (\boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{A}^T\boldsymbol{c}.$$

- Extract the estimation $\hat{\boldsymbol{q}}$ of the 3D position from the vector $\boldsymbol{x}$. The first three entries of $\boldsymbol{x}$ correspond to $\hat{\boldsymbol{q}}$.

- Calculate the points $\boldsymbol{q}_1$ and $\boldsymbol{q}_2$, which are closest to $\hat{\boldsymbol{q}}$, as

$$\boldsymbol{q}_1 = \boldsymbol{c}_1 + x_4\boldsymbol{r}_1 \quad \text{and} \quad \boldsymbol{q}_2 = \boldsymbol{c}_2 + x_5\boldsymbol{r}_2.$$

- Calculate the Root-Mean-Square-Error (RMSE) as

$$\text{RMSE} = \sqrt{\frac{1}{2}\left(\|\boldsymbol{q}_1 - \hat{\boldsymbol{q}}\|_2^2 + \|\boldsymbol{q}_2 - \hat{\boldsymbol{q}}\|_2^2\right)}$$

- Print the estimation $\hat{\boldsymbol{q}}$ of the 3D position and the RMSE.