# Python 3 for scientific computing

Lecture 1, 24.1.2018
Introduction and ~~30,000 ft~~ overview
9,144 m

Juha Jeronen
juha.jeronen@tut.fi

Spring 2018, TUT, Tampere
RAK-19006 Various Topics of Civil Engineering

TAMPERE UNIVERSITY OF TECHNOLOGY

# Meta

- Seminar on Python 3 from the viewpoint of scientific computing
- Timeslot: Wednesday, 12–14, in RO105
- **5 credits**; O(10) meetings
- Formal requirements to pass the course:
  - Final assignment: write a small, possibly useful project in Python (details will be announced later)
  - No exam; exercises almost surely useful, but not compulsory
- Lecture material: https://github.com/Technologicat/python-3-scicomp-intro/
- Also these slides will be available
- Don't worry, I will try to fix the course signup and any other practical issues.

# Python?

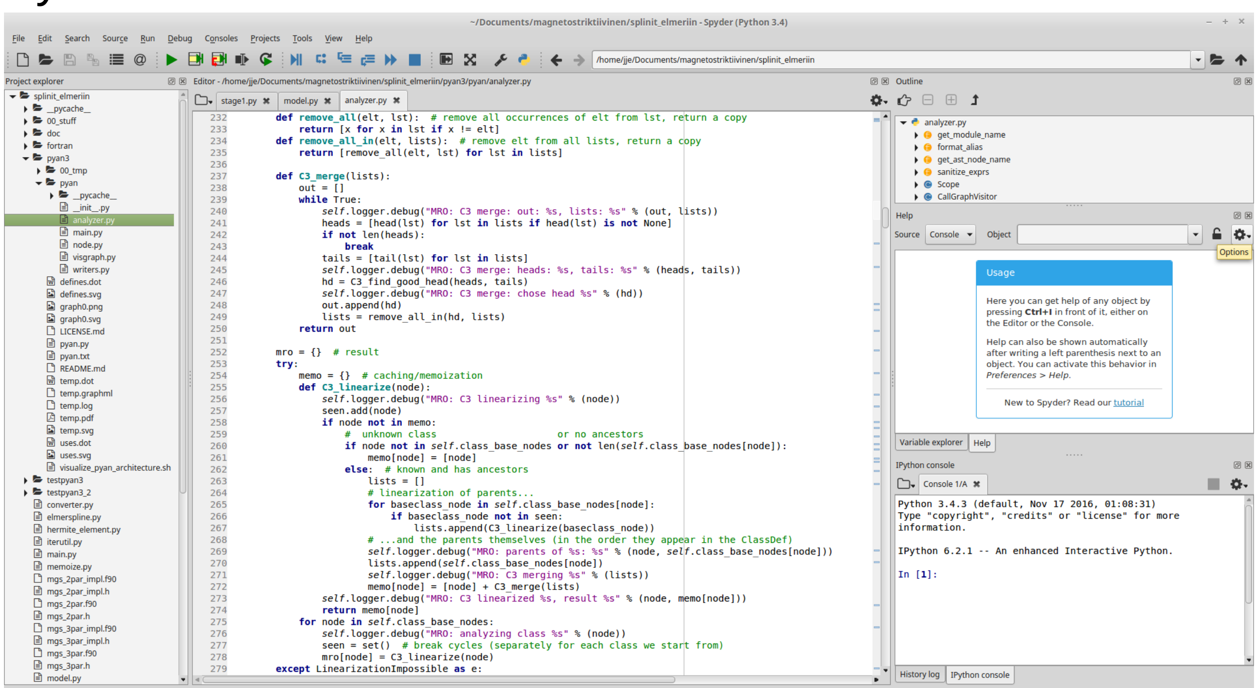
*Python natalensis*, Sir Andrew Smith, 1840.
https://commons.wikimedia.org/wiki/File:Python_natalensis_Smith_1840.jpg

A fairly long history:

- Python 1.0: *1991*
- Python 2.0: *2000*
- Python 3.0: *2008*

Family tree of languages:
https://www.levenez.com/lang/

Python 3:




https://www.python.org/

# What & why for numerics

- Clear, general-purpose, high-level, well-designed complete programming solution
- Easy to learn
- Focus on clarity: often Python programs look clear, making it easier to return to old code later
- Open source; repeatability and transparency of science
- Free of cost; no need for licenses
- “Complete” includes numerics; a viable competitor for MATLAB

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

print(“Hello, world!”)
```

https://www.python.org/

# Why now?

- Python 3 is a sufficiently stable platform to build science on
- Rise and popularization of Python during the last 15 years
- Python now part of the mainstream of programming: many libraries, extensive help available on the internet (especially StackOverflow)
- IEEE Spectrum 2017: Python the most popular language worldwide
  https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

f = lambda x: x**2
g = lambda x: x % 2 == 0
A = range(10)
B = [f(x) for x in A if not g(x)]
print(B)
```

https://www.python.org/

# Where MATLAB is good

- Quality-of-life features of a commercial product
  - All-in-one
  - Attempts to do some things automatically for the user, e.g. JIT: https://hips.seas.harvard.edu/blog/2013/05/13/jit-compilation-in-matlab/
- Community focused solely on numerics
  - If an algorithm is not in MATLAB, it can likely be found in MATLAB File Exchange
- Polished integrated development environment (IDE) for scientists
- Interactive plot editor
- Some things easier to do than in Python (e.g. 3D plotting)
- SimuLink

vs.

# Where Python is good

- Elegant language, in which it is easy to write clear code
- Control: does only what you explicitly tell it to
- Software ecosystem with dependency management (PyPI/pip)
  - A huge number of libraries for anything a computer can do
  - A sensibly sized numerics community, too
- Free of cost; no need for licenses
- Open; repeatability and transparency of science
  - In practice, also the libraries are open source. Sometimes a library already does 99% of what you need...

vs.

# “Python”

- Technically speaking, “*Python*” is a specification, like “*C*” or “*Fortran*”
    - Several implementations: *CPython*, *Jython*, *IronPython*, *PyPy*
    - *CPython* however de facto standard; for most == Python

- Python 3 vs. Python 2
    - Python 3: current version (2017, v3.6.3)
        - Also unofficially known as *py3k*, *Python 3000*
    - Python 2: legacy (2010, v2.7, support ends by 2020)
        - Python 2.x ends at 2.7: *Python 2.8 Un-release Schedule* https://www.python.org/dev/peps/pep-0404/
    - Practically all projects have already migrated to Python 3
    - This seminar: Python 3

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import re
print(re.sub(r'^monty\s*', r'', 'monty python') + " 3")
```

https://www.python.org/

# Python 2 vs. Python 3

?

Python 3?

- Almost the same language, but not quite
- Most significant change: Unicode support
  - Python was designed before Unicode, Unicode support was added on
  - Cleaned up in Python 3, but broke backward compatibility
- In some things Python 3 is a more elegant language, because it was cleaned up also in other respects in the major update
  - print() is a function; division / always outputs a float; integer division //; list comprehension has an internal scope
- Python core developer Nick Coghlan: we won't break backward compatibility again when the time for Python 4.0 comes
  http://www.curiousefficiency.org/posts/2014/08/python-4000.html
- Nick Coghlan: Python 3 Q&A:
  http://python-notes.curiousefficiency.org/en/latest/python3/questions_and_answers.html

# Python 3.x and PEP

- PEP = **P**ython **E**nhancement **P**roposal
- The main mechanism for suggesting extensions to the language
- An evolving language; some picks from recent accepted PEPs:
  - Literal string interpolation (f-strings) (3.6+), a new clear syntax for formatting strings
    https://www.python.org/dev/peps/pep-0498/
  - A dedicated infix operator for matrix multiplication (3.5+), A @ B
    https://www.python.org/dev/peps/pep-0465/
  - Additional unpacking generalizations (3.5+), an extension for the *tuple unpacking* syntax
    https://www.python.org/dev/peps/pep-0448/
  - Coroutines with async and await syntax (3.5+), decoupling of coroutines from generators (to avoid misleading programmer intuition)
    https://www.python.org/dev/peps/pep-0492/

# Tools

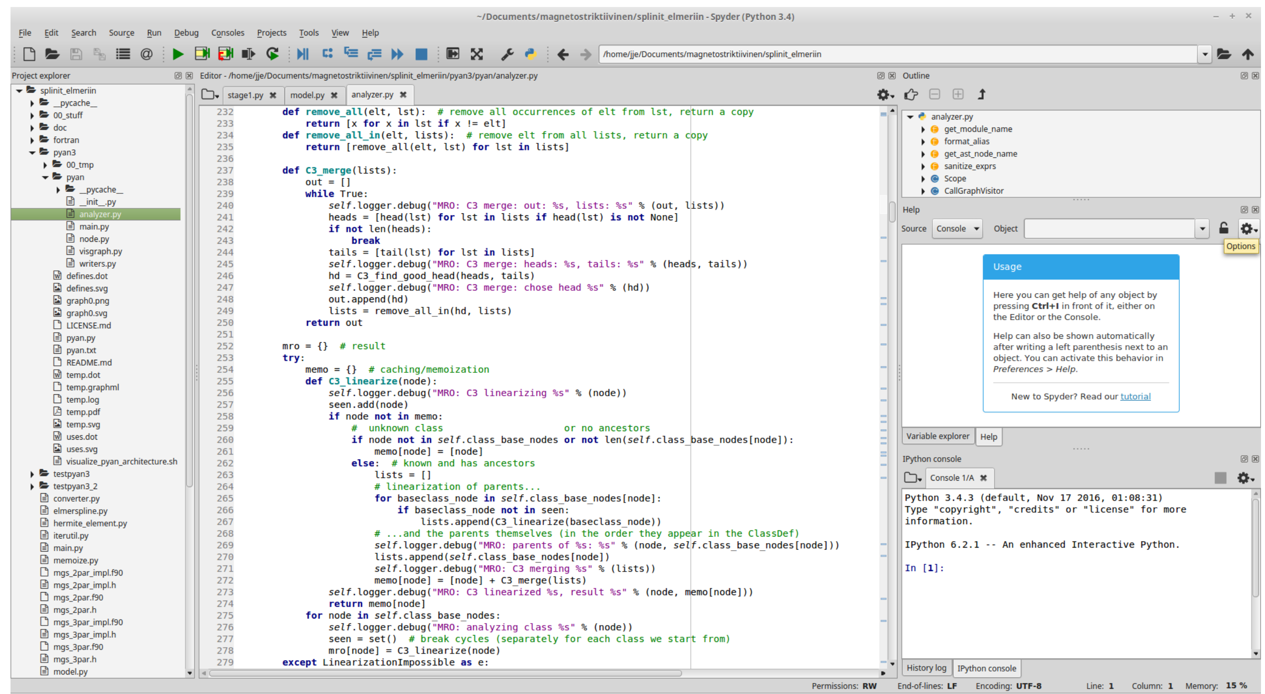
http://ipython.org/

https://www.anaconda.com/

https://github.com/spyder-ide

- **IPython**: advanced command line
- **Jupyter**: IPython's graphical cousin, based on a Mathematica-style *notebook* approach
- **Spyder**: integrated development environment (IDE)
- **Anaconda**: scientific Python distribution
  - Maybe the easiest way to install Python and its scientific libraries on a Windows computer.

# Spyder IDE




https://github.com/spyder-ide

- The **S**cientific **PY**thon **D**evelopment **E**nvi**R**onment
- Designed for scientists
- MATLAB style IDE
- Matplotlib integration
- Debugger
- Profiler
- Static code analyzer
- REPL (IPython/Jupyter)
  read-eval-print-loop

- Mostly well balanced between scientific use oriented, interactive, and software development oriented features.
- Cons: no automatic refactoring or version control GUI. Tools exist; a matter of implementing a frontend.

# Scientific Python

Welcome to the bazaar

**SciPy**
https://scipy.org/

**SymPy**
http://www.sympy.org/

**NumPy**
http://www.numpy.org/

**Matplotlib**
http://matplotlib.org/

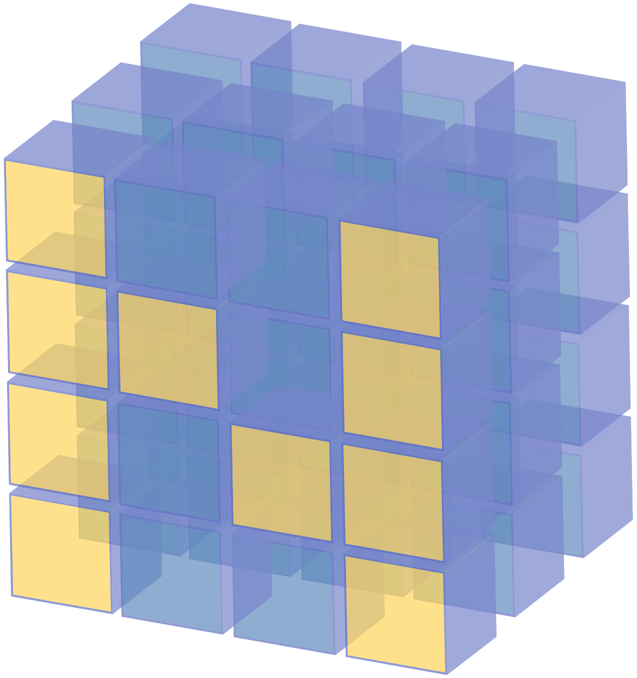
# Scientific Python

- Consists of separate libraries (as usual with general-purpose programming languages)
- **NumPy**, **SciPy**, **Matplotlib** the most important
  - These already do a lot
- **SymPy** for symbolic computing
  - We will look at this, too
- Some other, more specific libraries and their use cases are gathered in the lecture material (sec. 2)
  - We will also look at some of them later on the course

# NumPy

http://www.numpy.org/

- *n*-dimensional arrays
- MATLAB style API, but instead of matrices, based on *cartesian tensors:*
  - A vector is a rank 1 tensor
  - A matrix is a rank 2 tensor



```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import numpy as np
A = np.array( [[1, 2],
               [4, 9]], dtype=np.float64 )
b = np.array( [3, 7], dtype=np.float64 )
x = np.linalg.solve(A, b)
```

# SciPy

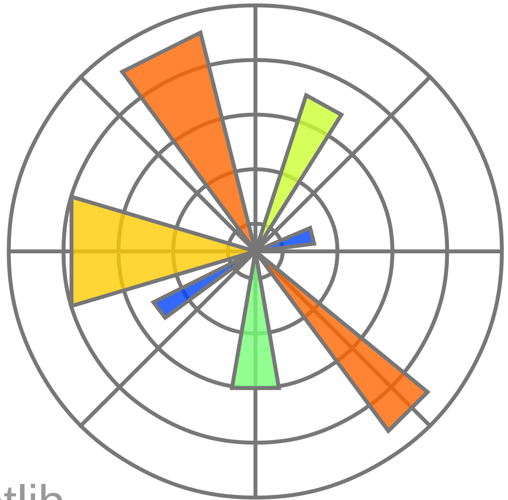
https://scipy.org/

- Advanced-user versions of linear algebra routines
- Sparse matrices
- I/O for MATLAB *.mat* files
- Numerical integration (quad), initial value problems (ODE), special functions
- Signal processing
- Some optimization routines
- Cython interface to LAPACK, for advanced users

# Matplotlib

http://matplotlib.org/

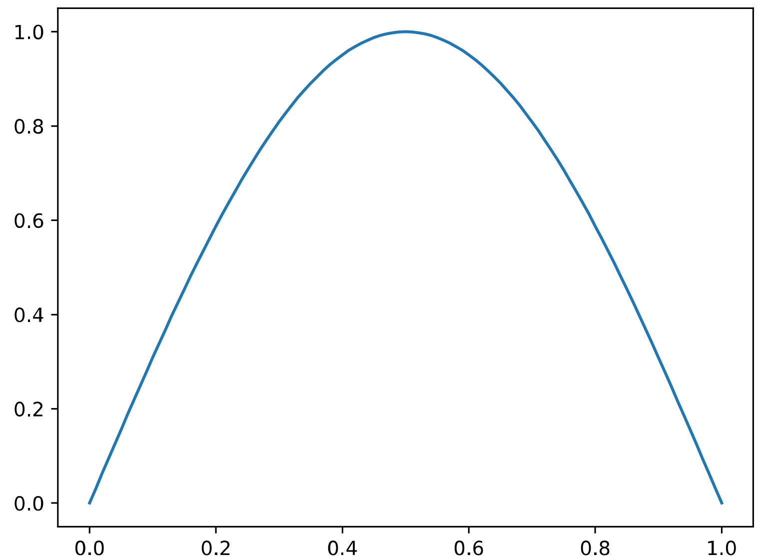
- MATLAB-style plotting API
- Standard tool for publication-quality numerical graphics in Python



Also made with Matplotlib

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import numpy as np
import matplotlib.pyplot as plt
xx = np.linspace(0, 1, 101)
yy = np.sin(xx * np.pi)
plt.plot(xx, yy)
plt.savefig("sin_x.svg")
```



# SymPy

http://www.sympy.org/

- Symbolic algebra, differentiation, integration

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import sympy as sy
x = sy.symbols('x')
λf,λg = sy.symbols('f,g', cls=sy.Function)
g = λg(x)
f = λf(g)
D = sy.diff(f, x).doit()
sy.pprint(D)
```

```
   d                   d
─────(f(g(x)))⋅──(g(x))
dg(x)              dx
```

- Python 3 allows Unicode variable names (with certain limitations).

Input e.g. using this:
https://github.com/clarkgrubb/latex-input

# Zen of Python, The

**import** this

https://www.python.org/dev/peps/pep-0020/

A guiding philosophy for the design of the Python language, as well as many Python programs.
Consists of 20 aphorisms, 19 of which have been written down:

*Beautiful is better than ugly.*
*Explicit is better than implicit.*
*Simple is better than complex.*
*Complex is better than complicated.*
*Flat is better than nested.*
*Sparse is better than dense.*
*Readability counts.*
*Special cases aren't special enough to break the rules.*
*Although practicality beats purity.*
*Errors should never pass silently.*
*Unless explicitly silenced.*
*In the face of ambiguity, refuse the temptation to guess.*
*There should be one– and preferably only one –obvious way to do it.*
*Although that way may not be obvious at first unless you're Dutch.*
*Now is better than never.*
*Although never is often better than **right** now.*
*If the implementation is hard to explain, it's a bad idea.*
*If the implementation is easy to explain, it may be a good idea.*
*Namespaces are one honking great idea – let's do more of those!*

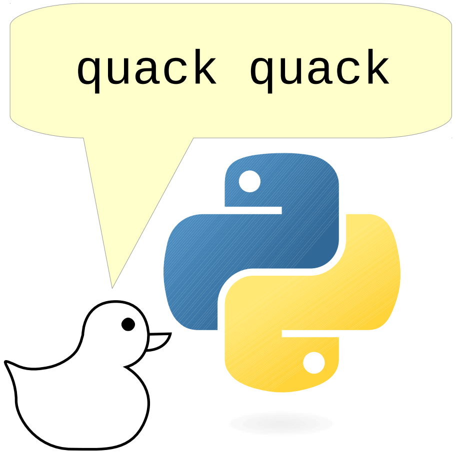
–Tim Peters

https://www.python.org/

# Python as a programming language

1) Imperative (but also some FP features)
2) Interpreted
3) Object-oriented
4) Lexically scoped
5) Duck typed
6) Call-by-sharing (call-by-object)
7) Reflective

functional programming




```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

class Duck:
    def __init__(self):
        pass
    def quack(self):
        print('quack quack')

d = Duck()
d.quack()
```

# Python as a programming language

## 1) Imperative, but also some FP features

- Imperative, like *C*, *C++*, *Fortran*, *Java*, *MATLAB*.
  - I.e. a program is a sequence of simple steps.
  - On the other hand, *imperative programming* has another connotation, especially in FP circles (e.g. *Haskell*, *Racket*):
  - Imperative programs contain *mutable state*, which is changed (*mutated*) as the computation proceeds; functional ones attempt to avoid this.
  - Cf. the thought of executable mathematics (L. Peter Deutsch) http://www.paulgraham.com/quotes.html

- Python can also be used for functional programming (FP):
  - Functions are *first-class objects* (can be passed as arguments, returned from functions, saved in a variable, and so on)
  - Anonymous functions, **lambda** x: ..., like MATLAB's @(x) …
  - Module functools in the standard library
  - High-level language; often, in the parts of the program that are not performance-critical, a functional approach is natural (no side effects ⇒ easier to test, fewer bugs).

# Python as a programming language

## 2) Interpreted

- Typical modern interpreted language. Compiled to bytecode, like *Java* and *MATLAB*.

- Bytecode runs on the Python virtual machine, contained in the Python interpreter.

- Bytecode can be disassembled. Possibly useful, if curious about how Python works:

```python
def f(x):
    return x**2
import dis
print(dis.dis(f))
```

# Python as a programming language

## 3) Object-oriented

- Everything is an object (also functions)
- Even the *types of objects* are objects
- At the top of the class hierarchy, everything inherits from *object*
- Multiple inheritance allowed
  - Method resolution order (MRO) determined by C3 linearization: https://en.wikipedia.org/wiki/C3_linearization
- No separate *struct*; a data structure with named fields ⇒ **class**
- Classes can be defined anywhere, also inside a function.
- Python is not as strict about object-orientedness as e.g. Java, where even the main() function must be a static method of a class. Python allows also classical functions (not part of any class).
- Python is not as strict about the procedural model either (cf. *C*). Even **main**() does not need to be defined, unless one specifically wants to.
- Like in *MATLAB*, any sequence of statements (and/or expressions) is a valid Python program.

# Python as a programming language

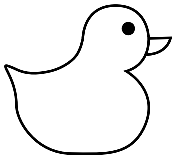
## 4) Lexically scoped

- Like in most modern languages.
- Lexical scope = a piece of the program's source code as text.
- A name declared in a certain lexical scope exists within the piece of text spanned by that scope, and nowhere else.
  (Confused? We will have examples/exercises about this.)
- In Python, the smallest unit of scope is the function.
  - A **for** loop, or most other blocks for that matter (e.g. **while**, **try**, **with**), do **not** introduce a local scope (contrast C++, Java).
  - List comprehension however does; it is processed like a function.
    - This was one of the things that was changed in Python 3.0.
- Name lookup follows the **LEGB** rule: **L**ocal, **E**nclosing, **G**lobal, **B**uilt-in.

https://en.wikipedia.org/wiki/Scope_(computer_science)#Lexical_scoping_vs._dynamic_scoping

# Python as a programming language

## 5) Duck typed



- *“If it quacks...”*

- *Any object may be used in any context, up until it is used in a way that it does not support.*
  https://en.wikipedia.org/wiki/Duck_typing

- No (typed) variables!
  - In Python, types are attached to *values* (object instances), which are referred to using untyped *names*.
  - Assignment re-assigns the name to point to the new value; it does **not** modify the original value.
  - *Ned Batchelder: Facts and myths about Python names and values*
    https://nedbatchelder.com/text/names.html

- Python is, however, both *dynamically typed* and *strongly typed.*
  - *Dynamically:* types are checked only at runtime (no compile-time checks).
  - *Strongly*: Types of values are never implicitly converted. Unlike in *Perl*, a string will not become a number even if it contains only digits.

# Python as a programming language

## 6) Call-by-sharing (call-by-object)

- Cf. the traditional approaches of *call-by-value* and *call-by-reference*.

- *Call-by-sharing* is neither of these!

- In Python, the caller and callee *share the same object instance*.

- Combined with how assignment works, this implies that mutable instances and immutable instances behave differently as function arguments.
  (We will have an exercise about this.)

- https://en.wikipedia.org/wiki/Evaluation_strategy#Call_by_sharing

# Python as a programming language

## 7) Reflective

- A running program may modify almost anything (including its own code), at any time.
  https://en.wikipedia.org/wiki/Reflection_(computer_programming)

- compile(), eval(), exec() – also possible to run code given or generated at run time.
  - **Note!** eval() for user-given input is a serious information security risk. Only use eval(), if it can be guaranteed that the input is harmless.

- Generally speaking, it is difficult or impossible to deduce anything about the behaviour of Python programs by static analysis approaches.
  - However, in practice, reflective features are used sparingly. Static code analyzers exist, and usually the results are good. (pyan, pyflakes, pylint)

# Python as a programming language

## +1) Other features

- 0-based indexing (contrast *MATLAB* and *Fortran*)
- Automatic memory management (garbage collection, GC)
- No need to declare names; assignment creates the name if nonexistent.
- *Expressions* vs. *statements*
  - Like in many imperative programming languages, starting with *Fortran 1* (1957).
  - An *expression* returns a value, a *statement* does not.
  - Explicit **return** statement required to return a value from a function. The default return value (if no **return**) is always the special value None.
- Named arguments.
  - Function arguments can be passed also by name, not only by position in the argument list. (More on this in the exercises.)
- Aggressive use of namespaces.
  - Usually, a quick look at the source code of a Python program is sufficient to tell which library and module each function comes from.

- *Indentation is part of the language syntax.*
  - Improves readability, and no need to explicitly terminate blocks.

# Meta: the next few weeks

- Weeks 2-3:
  - Putting the theory in practice: examples and exercises.
  - Exercises not compulsory, but almost surely useful.
- Week 4: Overview of the most important scientific libraries: NumPy, SciPy, Matplotlib, SymPy.
- Week 5: hands-on for NumPy, SciPy, Matplotlib, SymPy.

# Literature

- Mark Lutz: *Learning Python*, 5th ed., O'Reilly, 2013.
  - Standard “bible” of the trade, very comprehensive.
  - A couple of minor versions behind the latest Python.

- Luciano Ramalho: *Fluent Python: Clear, Concise and Effective Programming*, O'Reilly, 2015.
  - Python 3 for programmers coming from other languages. Focuses on features that are easily missed, if the reader is used to thinking in another programming language.

- Zed A. Shaw: *Learn Python 3 the Hard Way*, Addison–Wesley, 2017.
  - For newcomers to programming.

- Internet!
  - The lecture material lists some web links with each topic.

- Questions? ⊳ juha.jeronen@tut.fi