FEEG1201 Computing - Python for Engineering

Hans Fangohr

fangohr@soton.ac.uk

https://fangohr.github.io

@ProfCompMod@fosstodon.org

University of Southampton

Outline

Introduction Computing & Computational Engineering

Computing

- use of computers to support research and operation in science, engineering, industry and services
- applications include
  - analysis of data
  - data science / data analytics
  - artificial intelligence (AI) & machine learning (ML)
  - control
  - computer simulations
  - virtual design & optimisation

This course: Why Python?

- is relatively easy to learn [1]
- high efficiency: a few lines of code achieve a lot
- growing use in academia and industry, thus
- many relevant libraries available
- minimises the time of the programmer
- but: (naive) Python in general much slower execution than compiled languages (such as Fortran, C, C++, Rust, ...).

[1] https://link.springer.com/chapter/10.1007/978-3-540-25944-2_157

FEEG1201: Python computing

- introduces the foundations of Python programming language
- focus on parts of Python language and libraries relevant to engineering and design
- enable self-directed learning in the future

This course: practicalities

- 10 Lectures (this is lecture 1 [in teaching week 2])
- 9 computing laboratories (`lab1` to be discussed in tw 3)
    - sets of programming exercises
    - (automatic) feedback available
    - scheduled sessions
    - this is the key learning activity

First steps with Python

Hello World program

- Our first Python program: Entered interactively in Python prompt:

```
>>> print("Hello World")
Hello World
```

  Or in Interactive Python (IPython) prompt:

```
In [1]: print("Hello world")
Hello world
```

- Python prompt (>>>) and IPython prompt (`In [ ]:`) are very similar
- We prefer the more convenient IPython prompt (but the slides usually show the more compact >>> notation)

*Read-Eval-Print Loop (REPL)

The python and the IPython prompt are both examples for a READ-EVAL-PRINT LOOP (REPL):

- Read (the command the user enters)
- Evaluate (the command)
- Print (the result of the evaluation)
- Loop (i.e. go back to the beginning and wait for next command)

Integrated development environments (Spyder)

- You can write programs with a python prompt, a shell and an editor
- More convenient is the use of an “Integrated Development Environment” (IDE)
- Example IDEs: Spyder, Visual Studio Code, PyCharm, IDLE, Emacs, …
- A python prompt is typically embedded in the IDE
- We use Spyder in this module

Everything in Python is an object (with a type)

```
>>> type("Hello World")
<class 'str'>             # "Hello world" is a string
                          # 'class' means 'type'
>>> type(print)
<class 'builtin_function_or_method'>

>>> type(10)
<class 'int'>              # 10 is an integer number

>>> type(3.5)
<class 'float'>            # 3.5 is floating point number
                           # (floating point number: it has a decimal poi
>>> type('1.0')
<class 'str'>              # string (because of the quotes)

>>> type(1 + 3j)
<class 'complex'>          # complex number
```

Python prompt can act like a calculator

```
>>> 2 + 3
5
>>> 42 - 15.3
26.7
>>> 100 * 11
1100
>>> 2400 / 20
120
>>> 2 ** 3                    # 2 to the power of 3
8
>>> 9 ** 0.5                  # sqrt of 9
3.0
```

Create variables through assignment

```
>>> a = 10
>>> b = 20
>>> a              # short cut for 'print(a)'
10
>>> b              # short cut for 'print(b)'
20
>>> a + b          # ...
30
>>> ab4 = (a + b) / 4
>>> ab4
7.5
```

Functions

- Example: `print` function

```
>>> print("Hello World")
Hello World
```

The `print` function takes an argument (here a string), and does something with the argument. (Here printing the string to the screen.)

- Example: `abs` function

```
>>> x = -100
>>> y = abs(x)
>>> print(y)
100
```

A function may return a value: the `abs` function returns the absolute value (100) of the argument (-100).

The `help` function

The `help(x)` function provides documentation for object `x`.

Example:

```
>>> help(abs)
Help on built-in function abs in module builtins:

abs(x, /)
    Return the absolute value of the argument.
```

Summary useful commands (introspection)

- `print(x)` to display the object `x`

  Not needed at the prompt, but in programs that we will write later.

- `type(x)` to determine the type of object `x`
- `help(x)` to obtain the documentation string for object `x`
- To be introduced soon:

  `dir(x)` to display the methods and members of object `x`, or the current name space (`dir()`).

Functions

Defining a function ourselves

- Functions
  - provide (potentially complicated) functionality
  - are building blocks of computer programs
  - hide complexity from the user of the function
  - help manage complexity of software

- Example 1:

```python
def mysum(a, b):
    return a + b

# main program starts here
print("The sum of 3 and 4 is", mysum(3, 4))
```

Functions should be documented (“docstring”)

```python
def mysum(a, b):
    """Return the sum of parameters a and b."""
    return a + b

# main program starts here
print("The sum of 3 and 4 is", mysum(3, 4))
```

Can now use the help function for our new function:

```
>>> help(mysum)
Help on function mysum in module __main__:

mysum(a, b)
    Return the sum of parameters a and b.
```

Function documentation strings

```python
def mysum(a, b):
    """Return the sum of parameters a and b."""
    return a + b
```

Essential information for documentation string:

- What inputs does the function expect?
- What does the function do?
- What does it return?

*Desirable:

- Examples
- Notes on algorithm (if relevant)
- exceptions that might be raised
- [Author, date, contact details: not needed if version control is used]

LAB1

Advanced: Recommendations for documentation string style are numpydoc style or PEP257 docstring conventions.

Function documentation string example 1

```python
def mysum(a, b):
    """Return the sum of parameters a and b.

    Parameters
    ----------
    a : numeric
        first input
    b : numeric
        second input

    Returns
    -------
    a+b : numeric
        returns the sum (using the + operator) of a and b. The return type will
        depend on the types of `a` and `b`, and what the plus operator returns.

    Examples
    --------
    >>> mysum(10, 20)
    30
    >>> mysum(1.5, -4)
    -2.5

    Notes
    -----
    History: example first created 2002, last modified 2013
    Hans Fangohr, fangohr@soton.ac.uk,
    """
    return a + b
```

Function documentation string example 2

```python
def factorial(n):
    """Compute the factorial recursively.

    Parameters
    ----------
    n : int
        Natural number `n` > 0 for which the factorial is computed.

    Returns
    -------
    n! : int
        Returns n * (n-1) * (n-2) * ... * 2 * 1

    Examples
    --------
    >>> factorial(1)
    1
    >>> factorial(3)
    6
    >>> factorial(10)
    3628800
    """
    assert n > 0

    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

Function terminology

Example `abs(x)` function:

```
x = -1.5
y = abs(x)
```

- `x` is the *argument* given to the function (also called *input* or *parameter*)
- `y` is the *return value* (the result of the function's computation)
- Functions may expect zero, one or more arguments
- Not all functions (seem to) return a value. (If no `return` keyword is used, the special object `None` is returned.)

Function example

```
def plus42(n):
"""Add 42 to n and return""" # docstring

    result = n + 42          # body of
    return result            # function

# main program follows
a = 8
b = plus42(a)
```

After execution, `b` carries the value `50` (and a = 8).

Summary functions

- Functions provide (black boxes of) functionality:
  crucial building blocks that hide complexity
- interaction (input, output) through input arguments and return values

  (*printing* and *returning* values is not the same, see slide 29)
- docstring provides the specification (contract) of the function's input, output and behaviour
- a function should (normally) not modify input arguments

  (watch out for lists, dicts, more complex data structures as input arguments)

Functions printing vs returning values

Key message: functions should generally *return* values.

We use the Python prompt to explore the difference with these two function definitions:

```
def print42():
    print(42)

def return42():
    return 42
```

```
>>> b = return42()    # return 42, is assigned
>>> print(b)          # to b
42

>>> a = print42()     # return None, and
42                    # print 42 to screen
>>> print(a)
None                  # special object None
```

If we use IPython, it shows whether a function returns something (i.e. not None) through the `Out [ ]` token:

```
In [1]: return42()
Out[1]: 42              # Return value of 42


In [2]: print42()
42                      # No 'Out [ ]', so no
                        # returned value
```

Summary: to print or to return?

- A function that returns the control flow through the `return` keyword, will return the object given after `return`.
- A function that does not use the `return` keyword, returns the special object `None`.
- Generally, functions should return a value.
- Generally, functions should not print anything.
- Calling functions from the prompt can cause some confusion here: if the function returns a value and the value is not assigned, it will be printed.

About Python

Python

What is Python?

- High level programming language
- interpreted
- supports three main programming styles (imperative=procedural, object-oriented, functional)
- General purpose tool, yet good for numeric work with extension libraries

Availability

- Python is free
- Python is platform independent (works on Windows, Linux/Unix, Mac OS, ...)
- Python is open source

Python documentation

There is lots of documentation that you should learn to use:

- Teaching materials on website, including these slides and a text-book like document
    - Online documentation, for example
        - Python home page (http://www.python.org)
        - Matplotlib (publication figures)
        - Numpy (fast vectors and matrices, (NUMerical PYthon)
        - SciPy (scientific algorithms, `solve_ivp`)
        - SymPy (Symbolic calculation)
        - Pandas (wrangling and analysing tabular data)
- interactive documentation (`help()`)

Which Python version

- We use Python 3.
- For non-maintained software, Python 2.7 is still in use
- Python 2.x and 3.x are incompatible although the changes only affect very few commands.
- For this course, Python 3.10 or more recent is sufficient (3.12 preferred in August 2024).

Introspection (`dir`)

The directory function (`dir`)

- Everything in Python is an object.
- Python objects have *attributes*.
- `dir(x)` returns the attributes of object `x`
- Example:

```
>>> c = 2 + 1j
>>> dir(c)  # we ignore attributes starting with __
[ ... 'conjugate', 'imag', 'real']
>>> c.imag
1.0
>>> c.real
2.0
>>> c.conjugate()
(2-1j)
```

Attributes of objects can be functions

Example:

```
>>> c = 2 + 1j
>>> dir(c)
[ ... 'conjugate', 'imag', 'real']
>>> type(c.real)
<class 'float'>
>>> type(c.conjugate)
<class 'builtin_function_or_method'>
```

To *execute* a function, we need to add `()` to their name:

```
>>> c.conjugate    # this is the function object
<built-in method conjugate of complex object at 0x10a95f3d0>
>>> c.conjugate()  # this executes the function
(2-1j)             # return value of conjugate function
```

An object attribute that is a function, is called a *method*.

Introspection example with string

```
>>> word = 'test'
>>> print(word)
test
>>> type(word)
<class str>
>>> dir(word)
['__add__', '__class__', '__contains__', ...,
'__doc__', ..., 'capitalize', <snip>,
'endswith', ..., 'upper', 'zfill']
>>> word.upper()
'TEST'
>>> word.capitalize()
'Test'
>>> word.endswith('st')
True
>>> word.endswith('a')
False
```

Conditionals, if-else

Truth values

The python values `True` and `False` are special inbuilt objects:

```
>>> a = True
>>> print(a)
True
>>> type(a)
<class bool>
>>> b = False
>>> print(b)
False
>>> type(b)
<class bool>
```

We can operate with these two logical values using boolean logic, for example the logical and operation (`and`):

```
>>> True and True          # logical and operation
True
>>> True and False
False
>>> False and True
False
>>> False and False
False
```

There is also logical or (`or`) and the negation (`not`):

```
>>> True or False
True
>>> not True
False
>>> not False
True
>>> True and not False
True
```

In computer code, we often need to evaluate some expression that is either true or false (sometimes called a "predicate"). For example:

```
>>> x = 30        # assign 30 to x
>>> x >= 30       # is x greater than or equal to 30?
True
>>> x > 15        # is x greater than 15
True
>>> x > 30
False
>>> x == 30       # is x the same as 30?
True
>>> not x == 42   # is x not the same as 42?
True
>>> x != 42       # is x not the same as 42?
True
```

The `if-else` command allows to branch the execution path depending on a condition. For example:

```
>>> x = 30                # assign 30 to x
>>> if x > 30:            # predicate: is x > 30
...     print("Yes")          # if True, do this
... else:
...     print("No")           # if False, do this
...
No
```

The general structure of the `if-else` statement is

```
if A:
    B
else:
    C
```

where `A` is the predicate.

- If `A` evaluates to `True`, then all commands `B` are carried out (and `C` is skipped).
- If `A` evaluates to `False`, then all commands `C` are carried out (and `B`) is skipped.
- `if` and `else` are Python keywords.

`A` and `B` can each consist of multiple lines, and are grouped through indentation as usual in Python.

if-else example

```
def slength1(s):
    """Returns a string describing the
    length of the sequence s"""
    if len(s) > 10:
        ans = 'very long'
    else:
        ans = 'normal'

    return ans

>>> slength1("Hello")
'normal'
>>> slength1("HelloHello")
'normal'
>>> slength1("Hello again")
'very long'
```

if-elif-else example

If more cases need to be distinguished, we can use the keyword `elif` (standing for ELse IF) as many times as desired:

```
def slength2(s):
    if len(s) == 0:
        ans = 'empty'
    elif len(s) > 10:
        ans = 'very long'
    elif len(s) > 7:
        ans = 'normal'
    else:
        ans = 'short'

    return ans
```

```
>>> slength2("")
'empty'
>>> slength2("Good Morning")
'very long'
>>> slength2("Greetings")
'normal'
>>> slength2("Hi")
'short'
```

Style guide for Python code

Syntax versus style

- Python programs *must* follow Python syntax.
- Python programs *should* follow Python style guide, because
  - readability is key (debugging, documentation, team effort)
  - conventions improve effectiveness

Common style guide: PEP8

From http://www.python.org/dev/peps/pep-0008/:

- This style guide evolves over time as additional conventions are identified and past conventions are rendered obsolete by changes in the language itself.
- *"Readability counts"*: One of Guido van Rossum's key insights is that code is *read much more often than it is written*. The guidelines provided here are intended to improve the readability of code and make it consistent across the wide spectrum of Python code.

PEP8 Style guide

- Indentation: use 4 spaces
- One space around assignment operator (=) operator: `c = 5` and not `c=5`.
- Spaces around arithmetic operators can vary. Both `x = 3*a + 4*b` and `x = 3 * a + 4 * b` are okay.
- No space before and after parentheses: `x = sin(x)` but not `x = sin( x )`
- A space after comma: `range(5, 10)` and not `range(5,10)`.
- No whitespace at end of line
- No whitespace in empty line
- One or no empty line between statements within function

- Two empty lines between functions
- One import statement per line
- import first standard Python library (such as `math`), then third-party packages (`numpy`, `scipy`, ...), then our own modules
- no spaces around = when used in keyword arguments:
  `"Hello World".split(sep=' ')` but not
  `"Hello World".split(sep = ' ')`

PEP8 Style Summary

- Follow PEP8 guide, in particular for new code.
- Use tools to help us:
  - Spyder editor can show PEP8 violations (In Spyder 6: `Preferences` → `Completion and Linting` → `Code style and formatting` → `[X] Enable code style lintiing` → `[OK]`)
  - Similar tools/plugins are available for other editors. editors.
  - `pycodestyle` program available to check source code from command line (used to be called `pep8` in the past).
    To check file `myfile.py` for PEP8 compliance:

```
pycodestyle myfile.py
```

*Style conventions for *documentation strings*

- Python documentation strings (pydoc) conventions:
  - PEP257 docstring style (from 2001), basis for both
  - numpydoc style (science) and
  - Google pydoc style
- Examples on slide 23 and 24 are compatible with all conventions
- Editors can highlight deviations
- Program to check documentation string style compliance in file `myfile.py`:
  - `pydocstyle --convention=pep257 myfile.py`
  - `pydocstyle --convention=numpy myfile.py`
  - `pydocstyle --convention=google myfile.py`

Using modules

The math module (import math)

```
>>> import math
>>> math.sqrt(4)
2.0
>>> math.pi
3.141592653589793
>>> dir(math)       #attributes of 'math' object
['__doc__', '__file__', < snip >
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2',
'atanh', 'ceil', 'copysign', 'cos', 'e', 'erf',
'exp', <snip>, 'sqrt', 'tan', 'tanh', 'trunc']

>>> help(math.sqrt)       # ask for help on sqrt
sqrt(...)
    sqrt(x)
    Return the square root of x.
```

Name spaces and modules

Three (good) options to access a module:

1. use the full name:

```
import math
print(math.sin(0.5))
```

2. use some abbreviation

```
import math as m
print(m.sin(0.5))
print(m.pi)
```

3. import all objects we need explicitly

```
from math import sin, pi
print(sin(0.5))
print(pi)
```

Modules provide functionality

- each module provides some additional python functionality
- Python has many modules:
  - *Python Standard Library*: `math`, `pathlib`, `sys`, ...
  - Contributions from others: `numpy`, `jupyter`, `pytest`, ...
  - Every programmer can create their own modules.
- there is distinction between *module*, *package*, and *library* but in practice the terms are used interchangeably.

LAB2

Sequences

Sequences overview

Different types of sequences

- strings
- lists (mutable)
- tuples (immutable)
- arrays (mutable, part of numpy)

They share common behaviour.

Strings

```
>>> a = "Hello World"
>>> type(a)
<class str>
>>> len(a)
11
>>> print(a)
Hello World
```

Different possibilities to limit strings:

```
'A string'
"Another string"
"A string with a ' in the middle"
"""A string with triple quotes can
extend over several
lines"""
```

Strings 2 (exercise)

- Define a, b and c at the Python prompt:

```
>>> a = "One"
>>> b = "Two"
>>> c = "Three"
```

- Exercise: What do the following expressions evaluate to?

  1. `d = a + b + c`
  2. `5 * d`
  3. `d[0], d[1], d[2]` (indexing)
  4. `d[-1]`
  5. `d[4:]` (slicing)

Strings 3 (exercise)

```
>>> s="""My first look at Python was an
... accident, and I didn't much like what
... I saw at the time."""
```

For the string `s`:

- count the number of (i) letters '`e`' and (ii) substrings '`an`'
- replace all letters '`a`' with '`0`'
- make all letters uppercase
- make all capital letters lowercase, and all lower case letters to capitals

Lists

```
[]                           # the empty list
[42]                         # a 1-element list
[5, 'hello', 17.3]           # a 3-element list
[[1, 2], [3, 4], [5, 6]]     # a list of lists
```

- Lists store an ordered sequence of Python objects
- Access through index (and slicing) as for strings.
- use **`help()`**, often used list methods is **`append()`**

(In general computer science terminology, vector or array might be better name as the actual implementation is not a linked list, but direct $\mathcal{O}(1)$ access through the index is possible.)

Example program: using lists

```
>>> a = []                  # creates a list
>>> a.append('dog')         # appends string 'dog'
>>> a.append('cat')         # ...
>>> a.append('mouse')
>>> print(a)
['dog', 'cat', 'mouse']
>>> print(a[0])             # access first element
dog                         # (with index 0)
>>> print(a[1])             # ...
cat
>>> print(a[2])
mouse
>>> print(a[-1])            # access last element
mouse
>>> print(a[-2])            # second last
cat
```

Example program: lists containing a list

```
>>> a = ['dog', 'cat', 'mouse', [1, 10, 100, 1000]]
>>> a
['dog', 'cat', 'mouse', [1, 10, 100, 1000]]
>>> a[0]
dog
>>> a[3]
[1, 10, 100, 1000]
>>> max(a[3])
1000
>>> min(a[3])
1
>>> a[3][0]
1
>>> a[3][1]
10
>>> a[3][3]
1000
```

Sequences – more examples

```
>>> a = "hello world"
>>> a[4]
'o'
>>> a[4:7]
'o w'
>>> len(a)
11
>>> 'd' in a
True
>>> 'x' in a
False
>>> a + a
'hello worldhello world'
>>> 3 * a
'hello worldhello worldhello world'
```

Tuples

- tuples are very similar to lists
- tuples are *immutable* (unchangeable after they have been created) whereas lists are *mutable* (changeable)
- tuples are usually written using parentheses (↔ "round brackets"):

```
>>> t = (3, 4, 50)    # t for Tuple
>>> t
(3, 4, 50)
>>> type(t)
<class tuple>

>>> L = [3, 4, 50]    # compare with L for List
```

```
>>> L
[3, 4, 50]
>>> type(L)
<class list>
```

Tuples are defined by comma

- tuples are defined by the comma (!), not the parenthesis

```
>>> a = 10, 20, 30
>>> type(a)
<class tuple>
```

- the parentheses are usually optional (but should be written anyway):

```
>>> a = (10, 20, 30)
>>> type(a)
<class tuple>
```

Tuples are sequences

- normal indexing and slicing (because tuple is a sequence)

```
>>> t[1]
4
>>> t[:-1]
(3, 4)
```

Why do we need tuples (in addition to lists)?

1. use tuples if you want to make sure that a set of objects doesn't change.
2. Using tuples, we can assign several variables in one line (known as *tuple packing* and *unpacking*)

   ```
   x, y, z = 0, 0, 1
   ```

   This allows "instantaneous swap" of values:

   ```
       a, b = b, a
   ```

Strictly: "tuple packing" on right hand side and "sequence unpacking" on left.

3. functions return tuples if they return more than one object

```
def f(x):
    return x**2, x**3

a, b = f(x)
```

4. tuples can be used as keys for dictionaries as they are immutable

(Im)mutables

- Strings — like tuples — are immutable:

```
>>> a = 'hello world'          # String example
>>> a[3] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object does not support item assignment
```

- strings can only be ’changed’ by creating a new string, for example:

```
>>> a = a[0:3] + 'x' + a[4:]
>>> a
'helxo world'
```

Summary sequences

- lists, strings and tuples (and arrays) are sequences.
- sequences share the following operations

`a[i]`	returns element with index i of `a`
`a[i:j]`	returns elements i up to $j-1$
`len(a)`	returns number of elements in sequence
`min(a)`	returns smallest value in sequence
`max(a)`	returns largest value in sequence
`x in a`	returns `True` if `x` is element in `a`
`a + b`	concatenates `a` and `b`
`n * a`	creates `n` copies of sequence `a`

In the table above, `a` and `b` are sequences, `i`, `j` and `n` are integers, `x` is an element.

Conversions

- We can convert any sequence into a tuple using the `tuple` function:

```
>>> tuple([1, 4, "dog"])
(1, 4, 'dog')
```

- Similarly, the `list` function, converts sequences into lists:

```
>>> list((10, 20, 30))
[10, 20, 30]
```

- *Looking ahead* to iterators, we note that `list` and `tuple` can also convert from iterators:

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

- *And if you ever need to create an iterator from a sequence, the `iter` function can this:

```
>>> iter([1, 2, 3])
<list_iterator object at 0x1013f1fd0>
```

Loops

Introduction loops

Computers are good at repeating tasks (often the same task for many different sets of data).

Loops are the way to execute the same (or very similar) tasks repeatedly ("in a loop").

Python provides the "for loop" and the "while loop".

Example program: for-loop

```
animals = ['dog', 'cat', 'mouse']

for animal in animals:
    print(f"This is the {animal}!")
```

produces

```
This is the dog!
This is the cat!
This is the mouse!
```

The for-loop *iterates* through the sequence `animals` and assigns the values in the sequence subsequently to the name `animal`.

Iterating over integers

Often we need to iterate over a sequence of integers:

```
for i in [0, 1, 2, 3, 4, 5]:
    print(f"the square of {i} is {i**2}")
```

produces

```
the square of 0 is 0
the square of 1 is 1
the square of 2 is 4
the square of 3 is 9
the square of 4 is 16
the square of 5 is 25
```

Iterating over integers with `range`

The `range(n)` object is used to iterate over a sequence of increasing integer values up to (but not including) `n`:

```
for i in range(6):
    print(f"the square of {i} is {i**2}")
```

produces

```
the square of 0 is 0
the square of 1 is 1
the square of 2 is 4
the square of 3 is 9
the square of 4 is 16
the square of 5 is 25
```

The range object

- `range` is used to iterate over integer sequences
- We can use the range object in for loops:

```
>>> for i in range(3):
...     print(f"i={i}")
i=0
i=1
i=2
```

- We can convert it to a list:

```
>>> list(range(6))
[0, 1, 2, 3, 4, 5]
```

- This conversion to list is useful to understand what sequences the range object would provide if used in a for loop:

```
>>> list(range(6))
[0, 1, 2, 3, 4, 5]
>>> list(range(0, 6))
[0, 1, 2, 3, 4, 5]
>>> list(range(3, 6))
[3, 4, 5]
>>> list(range(-3, 0))
[-3, -2, -1]
```

- *Advanced: `range` has its own type:

```
>>> type(range(6))
<class range>
```

`range` objects are lazy sequences (Python range is not an iterator)

Summary `range`

`range`

`range([start,] stop [,step])` iterates over integers from `start` up to to `stop` (*but not including* `stop`) in steps of `step`.

`start` defaults to 0 and `step` defaults to 1.

```
>>> list(range(0, 10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(0, 10, 2))
[0, 2, 4, 6, 8]
>>> list(range(5, 4))
[]                          # no iterations
```

Iterating over sequences with `for`-loop

- `for` loop iterates over iterables.
- Sequences are iterable.
- Examples

```python
for i in [0, 3, 4, 19]:          # list is a
    print(i)                     # sequence

for animal in ['dog', 'cat', 'mouse']:
    print(animal)

for letter in "Hello World":     # strings are
    print(letter)                # sequences

for i in range(5):               # range objects
    print(i)                     # are iterable
```

Example: create list with for-loop

```
def create_list_of_increasing_halfs(n):
    """Given integer n >=0, return list of length
    n starting with [0, 0.5, 1.0, 1.5, ...]."""
    result = []
    for i in range(n):
        number = i * 1 / 2
        result.append(number)
    return result

# main program
print(create_list_of_increasing_halfs(5))
```

Output:

```
[0.0, 0.5, 1.0, 1.5, 2.0]
```

Example: modify list with for-loop

```python
def modify_list_add_42(original_list):
    """Given a list, add 42 to every element
    and return"""
    modified_list = []
    for element in original_list:
        new_element = element + 42
        modified_list.append(new_element)
    return modified_list

# main program
print(modify_list_add_42([0, 10, 100, 1000]))
```

Output:

```
[42, 52, 142, 1042]
```

Reminder: If-then-else

- Example 1 (if-then-else)

```python
a = 42
if a > 0:
    print("a is positive")
else:
    print("a is negative or zero")
```

Another iteration example

This example generates a list of numbers often used in hotels to label floors (more info)

```
def skip13(a, b):
    """Given ints a and b, return
    list of ints from a to b without 13"""
    result = []
    for k in range(a, b):
        if k == 13:
            pass  # do nothing
        else:
            result.append(k)
    return result
```

Another iteration example (with `continue`)

This example generates a list of numbers often used in hotels to label floors (more info)

```
def skip13(a, b):
    """Given ints a and b, return
    list of ints from a to b without 13"""
    result = []
    for k in range(a, b):
        if k == 13:
            continue  # jump to next iteration
        result.append(k)
    return result
```

Exercise range_double

Write a function `range_double(n)` that generates a list of numbers similar to `list(range(n))`. In contrast to `list(range(n))`, each value in the list should be multiplied by 2. For example:

```
>>> range_double(4)
[0, 2, 4, 6]
>>> range_double(10)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

For comparison the behaviour of `range`:

```
>>> list(range(4))
[0, 1, 2, 3]
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

LAB3

For loop summary

- `for`-loop to iterate over sequences
- can use `range` to generate sequences of integers
- special keywords:
    - `continue` - skip remainder of body of statements and continue with next iteration
    - `break` - leave for-loop immediately
- *Advanced:
    - can iterate over any *iterable*
    - we can create our own iterables
    - See summary Socratica on Iterators, Iterables, and Itertools

Exercise: First In First Out (FIFO) queue

Write a *First-In-First-Out* queue implementation, with functions:

- `add(name)` to add a customer with name `name` (call this when a new customer arrives)
- `next()` to be called when the next customer will be served. This function returns the name of the customer
- `show()` to print all names of customers that are currently waiting
- `length()` to return the number of currently waiting customers

Suggest to use a global variable `q` and define this in the first line of the file by assigning an empty list: `q = []`.

While loops

- Reminder:
  a `for` loop iterates over a given sequence or iterator
- A `while` loop iterates *while a condition is fulfilled*
- ```
  x = 64
  while x > 10:
      x = x // 2
      print(x)
  ```
  produces
  ```
  32
  16
  8
  ```
  ```

*While loop example 2

Determine ϵ:

```
eps = 1.0

while eps + 1 > 1:
    eps = eps / 2.0
print(f"epsilon is {eps}")
```

Output:

```
epsilon is 1.11022302463e-16
```

*Iterables and iterators

- an object is *iterable* if the for-loop can iterate over it
- an *iterator* has a `__next()__` method, i.e. can be used with `next()`. The iterator is iterable.

```
>>> i = iter(["dog", "cat"])     # create iterator
                                 # from list
>>> next(i)
'dog'
>>> next(i)
'cat'
>>> next(i)                      # reached end
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  StopIteration
```

*Generators

- Generators are functions defined using `yield` instead of `return`
- When called, a generator returns an *object that behaves like an iterator*: it has a `next` method.

```
>>> def squares(n):
...     for i in range(n):
...         yield i**2
...
>>> s = squares(3)
>>> next(s)
0
```

```
>>> next(s)
1
>>> next(s)
4
>>> next(s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

The execution flow returns at the `yield` keyword (similar to return), but the flow continues after the `yield` when the `next` method is called the next time.

A more detailed example demonstrates this:

```python
def squares(n):
    print("begin squares()")
    for i in range(n):
        print(f"  before yield i={i}")
        yield i**2
        print(f"  after yield i={i}")

>>> g = squares(3)
>>> next(g)
begin squares()
  before yield i= 0
0
>>> next(g)
  after yield i= 0
  before yield i= 1
```

```
1
>>> next(g)
  after yield i= 1
  before yield i= 2
4
>>> next(g)
  after yield i= 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

See also Socratica on Iterators, Iterables, and Itertools

Reading and writing files

File input/output

It is a common task to

- read some input data file
- do some calculation/filtering/processing with the data
- write some output data file with results

Python distinguishes between

- *text* files (`'t'`)
- *binary* files `'b'`)

If we don't specify the file type, Python assumes we mean text files.

Writing a text file

```
>>> with open('test.txt', 'tw') as f:
...     f.write("first line\nsecond line")
...
22
```

creates a file `test.txt` that reads

```
first line
second line
```

- To write data, we need to open the file with `'w'` mode:

```
with open('test.txt', 'w') as f:
```

  By default, Python assumes we mean text files. However, we can be explicit and say that we want to create a Text file for Writing:

```
with open('test.txt', 'wt') as f:
```

- If the file exists, it will be overridden with an empty file when the open command is executed.
- The file object `f` has a method `f.write` which takes a string as an input argument.

Reading a text file

We create a file object `f` using

```
>>> with open('test.txt', 'rt') as f:   # Read Text
```

and have different ways of reading the data:

1. `f.read()` returns one long string for the whole file

   ```
   >>> with open('test.txt', 'rt') as f:
   ...     data = f.read()
   ...
   >>> data
   'first line\nsecond line'
   ```

2. `f.readlines()` returns a list of strings (each being one line)

```
>>> with open('test.txt', 'rt') as f:
...     lines = f.readlines()
...
>>> lines
['first line\n', 'second line']
```

3. *Advanced: Use text file `f` as an iterable object: process one line in each iteration

```
>>> with open('test.txt', 'rt') as f:
>>>     for line in f:
...         print(line, end='')
...
first line
second line
>>> f.close()
```

This is important for large files: the file can be larger than the computer RAM as only one line at a time is read from disk to memory.

*File input and output without context manager

With file context manager (recommended):

```
>>> with open('test.txt', 'rt') as f:   # This creates
...                                     # the context.
...     data = f.read()                 # We can use 'f'
...                                     # in the context.
...                    # File 'f' is automatically closed
>>> data               # when the context is left.
'first line\nsecond line'
```

Without file context manager (not recommended!):

```
>>> f = open('test.txt', 'rt')
>>> data = f.read()
>>> f.close()  # must close file manually
>>> data
'first line\nsecond line'
```

Use case: Reading a file, iterating over lines

Often we want to process line by line. Typical code fragment:

```
with open('myfile.txt', 'rt') as f:
    lines = f.readlines()

# some processing of the lines object
for line in lines:
    print(line)
```

Splitting a string

- We often need to split a string into smaller parts: use string method `split()`:
  (try `help("".split)` at the Python prompt for more info)

Example:

```
>>> c = 'This is my string'
>>> c.split()
['This', 'is', 'my', 'string']
>>> c.split('i')
['Th', 's ', 's my str', 'ng']
```

Useful functions processing text files:

- `string.strip()` method gets rid of leading and trailing white space, i.e. spaces, newlines (`\n`) and tabs (`\t`):

```
>>> a = "   hello\n "
>>> a.strip()
'hello'
```

- `int()` and `float` convert strings into numbers (if possible)

```
>>> int("42")
42
>>> float("3.14")
3.14
>>> int("0.5")
Traceback (most recent call last):
  ValueError: invalid literal for int()
            with base 10: '0.5'
```

Exercise: Shopping list

Given a list

```
bread       1  1.39
tomatoes    6  0.26
milk        3  1.45
coffee      3  2.99
```

Write program that computes total cost per item, and writes to `shopping_cost.txt`:

```
bread       1.39
tomatoes    1.56
milk        4.35
coffee      8.97
```

One solution

One solution is `shopping_cost.py`

```
with open('shopping.txt', 'tr') as fin:        # INput File
    lines = fin.readlines()

with open('shopping_cost.txt', 'tw') as fout: # OUTput File
    for line in lines:
        words = line.split()
        itemname = words[0]
        number = int(words[1])
        cost = float(words[2])
        totalcost = number * cost
        fout.write(f"{itemname:10} {totalcost}\n")
```

Exercise

Write function `print_line_sum_of_file(filename)` that reads a file of name `filename` containing numbers separated by spaces, and which computes and prints the sum for each line. A data file might look like

```
2 3 5 -30 100
0 45 3 2
17
```

LAB4

*Binary files 1

- Files that store *binary* data are opened using the `'b'` flag (instead of `'t'` for Text):

```
open('data.dat', 'br')
```

- For text files, we read and write `str` objects. For binary files, use the `bytes` type instead.
- By default, store data in text files. Text files are human readable (that's good) but take more disk space than binary files.
- Reading and writing binary data is outside the scope of this introductory module. If you read arbitrary binary data, you may need the `struct` module.
- For large/complex scientific data, consider HDF5.

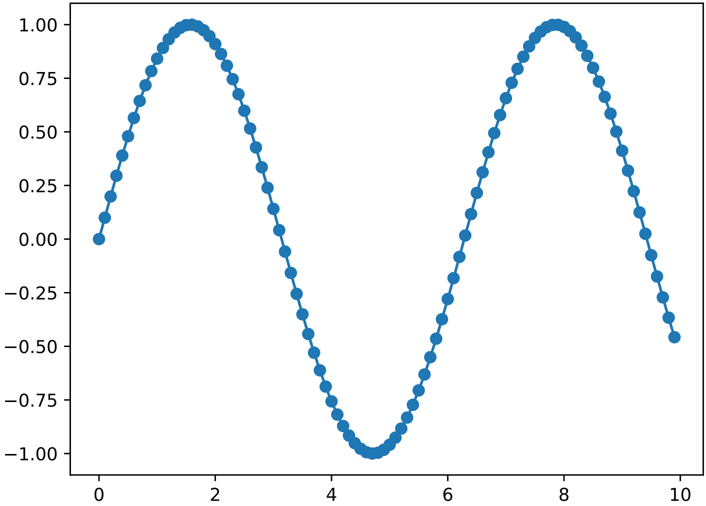
*HDF5 files

- If you need to store large and/or complex data, consider the use of HDF5 files:
  `https://portal.hdfgroup.org/display/HDF5/HDF5`
- Python interface: `https://www.h5py.org` (`import h5py`)
- hdf5 files
  - provide a hierarchical structure (like subdirectories and files)
  - can compress data on the fly
  - supported by many tools
  - standard in some areas of science
  - optimised for large volume of data and effective access

Outlook: first plot

```
import math
import matplotlib.pyplot as plt  # convention

xs = []  # store x-values for plot in list
ys = []  # store y-values for plot in list
for i in range(100):  # compute data
    x = 0.1 * i
    xs.append(x)
    y = math.sin(x)  # we plot sin(x)
    ys.append(y)

# plot data
plt.plot(xs, ys, '-o')

plt.savefig("matplotlib-mini-example.pdf")
```

Outlook: first plot



str, repr and eval

The `str` function and `__str__` method

All objects in Python should provide a method `__str__` which returns an *informal* string representation of the object.

This method `a.__str__` is called when we apply the `str` function to object `a`:

```
>>> a = 3.14
>>> a.__str__()
'3.14'
>>> str(a)
'3.14'
```

```
>>> import datetime
>>> now = datetime.datetime.now()
>>> now
datetime.datetime(2022, 1, 13, 13, 44, 56, 392268)
>>> str(now)
'2022-01-13 13:44:56.392268'
```

Implicit calling of `str` function

The string method `x.__str__` of object `x` is called implicitly, when we

- pass the object `x` directly to the print command
- use the "`{x}`" notation in f-strings

```
>>> now = datetime.datetime.now()
>>> now
datetime.datetime(2022, 1, 13, 13, 44, 56, 392268)
>>> print(now)
2022-01-13 13:44:56.392268
>>> f"{now}"
'2022-01-13 13:44:56.392268'
```

*The `repr` function and `__repr__` method

- The `repr` function should convert a given object into an *as accurate as possible* string representation
- The `repr` function will generally provide a more detailed string than `str`.
- Applying `repr` to the object `x` will attempt to call `x.__repr__()`.
- The python (and IPython) prompt uses `repr` to 'display' objects.

Example:

```
>>> import datetime
>>> t = datetime.datetime.now()
>>> str(t)
'2022-01-13 13:55:39.158456'
>>> repr(t)
'datetime.datetime(2022, 1, 13, 13, 55, 39, 158456)'
>>> t
datetime.datetime(2022, 1, 13, 13, 55, 39, 158456)
```

For many objects, `str(x)` and `repr(x)` return the same string.

*The `eval` function

The `eval` function accepts a string, and *evaluates* the string (as if it was entered at the Python prompt):

```
>>> x = 1
>>> eval('x + 1')
2
>>> s = "[10, 20, 30]"
>>> type(s)
<class str>
>>> eval(s)
[10, 20, 30]
>>> type(eval(s))
<class list>
```

*The repr and eval function

Given an accurate representation of an object as a string, we can convert that string into the object using the eval function.

```
>>> i = 42
>>> type(i)
<class int>
>>> repr(i)
'42'
>>> type(repr(i))
<class str>
>>> eval(repr(i))
42
>>> type(eval(repr(i)))
<class int>
```

The `datetime` example:

```
>>> import datetime
>>> t = datetime.datetime.now()
>>> t_as_string = repr(t)
>>> t_as_string
'datetime.datetime(2016, 9, 8, 14, 28, 48, 648192)'
>>> t2 = eval(t_as_string)
>>> t2
datetime.datetime(2016, 9, 8, 14, 28, 48, 648192)
>>> type(t2)
<class datetime.datetime>
>>> t == t2
True
```

Print

`print` function

- the `print` function sends content to the “standard output” (usually the screen)
- `print()` prints an empty line:

```
>>> print()
```

- Given a single string argument, this is printed, followed by a new line character:

```
>>> print("Hello")
Hello
```

- Given another object (not a string), the `print` function will *ask* the object for its preferred way to be represented as a string (via the `__str__` method):

```
>>> print(42)
42
```

- Given multiple objects separated by commas, they will be printed separated by a space character:

```
>>> print("dog", "cat", 42)
dog cat 42
```

- To supress printing of a new line, use the `end` option:

```
>>> print("Dog", end=""); print("Cat")
DogCat
>>>
```

Common strategy for the print command

- Construct some string `s`, then print this string using the `print` function

```
>>> s = "I am the string to be printed"
>>> print(s)
I am the string to be printed
```

- The question is, how can we construct the string `s`? We talk about string formatting.

String formatting

String formatting & Example 1

- Task: Given some objects, we would like to create a string representation.
- Example 1: a variable `t` has the value 42.123 and we like to print `Duration is 42.123s` to the screen.
- Solution: Create a *formatted string* “`Duration is 42.123s`” and pass this string to the `print` function:

```
>>> t = 42.123
>>> print(f"Duration = {t}s")
Duration = 42.123s
```

- With *string formatting*, we mean the creation of the string “`Duration is 42.123s`”

String formatting & Example 2

- Example 2: a variable `t` has the value 42.123 and we like to print `Duration is 42.1s` to the screen (i.e round to one post-decimal digit.)
- Solution:

```
>>> t = 42.123
>>> print(f"Duration = {t:.1f}s")
Duration = 42.1s
```

String formatting: Example 2 explanation

Explanation of `f"Duration = {t:.1f}s"`

`f"`	Beginning of a *f*ormatted string literal
`Duration =`	string content
`{…}`	content in curly braces is evaluated by Python
`t`	take value from variable t
`:f`	format t as a floating point number
`.1`	display one digit after the decimal point
`s`	string content
`"`	end of formatted string literal

String formatting examples with numbers

```
>>> import math
>>> p = math.pi
>>> f"{p}"  # default representation (same as `str(p)`)
'3.141592653589793'
>>> str(p)
'3.141592653589793'
>>> f"{p:f}"  # as floating point number (6 post-dec digits)
'3.141593'
>>> f"{p:10f}"  # total number 10 characters wide
'  3.141593'
>>> f"{p:10.2f}"  # 10 wide and 2 post-decimal digits
'      3.14'
>>> f"{p:.10f}"  # 10 post-decimal digits
'3.1415926536'
>>> f"{p:e}"  # in exponential notation
'3.141593e+00'
>>> f"{p:g}"  # extra compact
'3.14159'
```

Expressions in f-strings are evaluated at run time

We can evaluate Python expressions in the f-strings:

```
>>> import math
>>> f"The diagonal has length {math.sqrt(2)}."
'The diagonal has length 1.4142135623730951.'
```

---

*Advanced: Precision specifier can also be variables:

```
>>> width = 10
>>> precision = 4
>>> f"{math.pi:{width}.{precision}}"
'     3.142'
```

Show variable name and value with {name=}

Convenient short cut for debugging print statements:

```
>>> a = 10
>>> b = 20
>>> c = math.sqrt(a**2 + b**2)
>>> f"State: {a=} {b=} {c=}"
'State: a=10 b=20 c=22.360679774997898'
```

String formatting method overview

“f-strings”: most convenient and recommended method (2016):

```
>>> value = 42
>>> f"the value is {value}"
'the value is 42'
```

“new style” or “advanced” string formatting (Python 3, 2006):

```
>>> "the value is {}".format(value)
'the value is 42'
```

“% operator” (Python 1 and 2):

```
>>> "the value is %s" % value
'the value is 42'
```

Default function arguments

Default argument values for functions

- Motivation:
  - suppose we need to compute the area of rectangles and
  - we know the side lengths `a` and `b`.
  - Most of the time, `b=1` but sometimes `b` can take other values.
- Solution 1:

```python
def area(a, b):
    return a * b

print(f"The area is {area(3, 1)}")
print(f"The area is {area(2.5, 1)}")
print(f"The area is {area(2.5, 2)}")
```

- We can make the function more user friendly by providing a *default* value for `b`. We then only have to specify `b` if it is different from this default value:
- Solution 2 (with default value for argument `b`):

```
def area(a, b=1):
    return a * b

print(f"The area is {area(3)}")
print(f"The area is {area(2.5)}")
print(f"The area is {area(2.5, 2)}")
```

- Default parameters *have to be at the end* of the argument list in the function definition.

Default argument values

You may have met default arguments in use before, for example

- the `print` function uses `end='\n'` as a default value
- the `open` function uses `mode='rt'` as a default value
- the `list.pop` method uses `index=-1` as a default

LAB6

Keyword function arguments

Keyword argument values

- We can call functions with a “keyword” and a value. (The keyword is the name of the variable in the function definition.)
- Here is an example

```
def f(a, b, c):
    print(f"{a=} {b=} {c=}")

f(1, 2, 3)
f(c=3, a=1, b=2)
f(1, c=3, b=2)
```

which produces this output:

```
a=1 b=2 c=3
a=1 b=2 c=3
a=1 b=2 c=3
```

- If we use *only* keyword arguments in the function call, then we do not need to know the *order* of the arguments. (This is good.)
- Choosing meaningful variable names in the function definition makes the function more user friendly.

*Disallow or enforce keyword argument use

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
      |                |                  |
      |        Positional or keyword      |
      |                                   - Keyword only
       -- Positional only
```

See https://www.python.org/dev/peps/pep-0570/#how-to-teach-this

```
def standard_arg(arg):
    print(arg)

def pos_only_arg(arg, /):
    print(arg)
```

```python
def kwd_only_arg(*, arg):
    print(arg)

def combined_example(pos_only, /, standard, *, kwd_only):
    print(pos_only, standard, kwd_only)
```

List comprehension

List comprehension - one slide summary

```
>>> xs = [2*i for i in range(5)]   # 'list comprehension'
>>> print(xs)
[0, 2, 4, 6, 8]
```

is equivalent to this for set of commands with a for loop:

```
>>> xs = []
>>> for i in range(5):
...     xs.append(2*i)
...
>>> print(xs)
[0, 2, 4, 6, 8]
```

- useful when we need to process or create a list quickly
- no additional functionality over for-loop
- sometimes more elegant (≈ shorter) than for-loop

List comprehension

- List comprehension follows the mathematical “set builder notation”
- Convenient way to process a list into another list (without for-loop).

Examples

```
>>> [2*i for i in range(5)]
[0, 2, 4, 6, 8]

>>> [x**2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

List comprehension structure

Structure of list comprehension:

```
[EXPRESSION(OBJECT) for OBJECT in SEQUENCE]
```

where `EXPRESSION`, `OBJECT`, and `SEQUENCE` can vary.

Examples:

```
>>> [2*i for i in range(5)]
[0, 2, 4, 6, 8]

>>> import math
>>> [math.sqrt(x) for x in [1, 4, 9, 16]]
[1.0, 2.0, 3.0, 4.0]

>>> [s.capitalize() for s in ["dog", "cat", "mouse"]]
['Dog', 'Cat', 'Mouse']
```

List comprehension example 1 and 2

Can be useful to populate lists with numbers quickly

- Example 1:

```
>>> ys = [x**2 for x in range(10)]
>>> ys
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- Example 2:

```
>>> import math
>>> xs = [0.1 * i for i in range(5)]
>>> xs
[0.0, 0.1, 0.2, 0.3, 0.4]
>>> ys = [math.exp(x) for x in xs]
>>> ys
[1.0, 1.1051709180756477, 1.2214027581601699,
 1.3498588075760032, 1.4918246976412703]
```

List comprehension with filter

```
[EXPRESSION(OBJECT) for OBJECT in SEQUENCE
                        if CONDITION(OBJECT)]
```

- include `OBJECT` only if `CONDITION(OBJECT)` is `True`.
- Example:

```
>>> [i for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> [i for i in range(10) if i > 5]
[6, 7, 8, 9]

>>> [i for i in range(10) if i**2 > 5]
[3, 4, 5, 6, 7, 8, 9]
```

*Dictionary comprehension

In addition to *list comprehension* there is also *dictionary comprehension* available:

```
>>> {x: x**2 for x in range(5)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

>>> {word: len(word) for word in ["dog", "bird", "mouse"]}
{'dog': 3, 'bird': 4, 'mouse': 5}
```

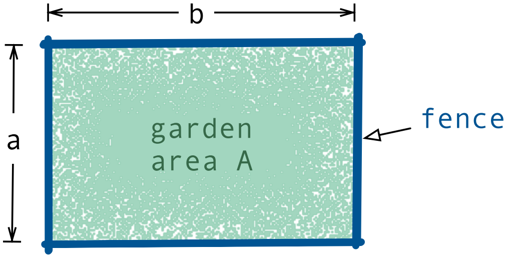
*Generator comprehension (advanced)

Generators (see slide 92) can also be created using a comprehension syntax:

```
>>> gen = (x**2 for x in range(5))
>>> type(gen)
<class 'generator'>
>>> for item in gen:
...     print(item)
...
0
1
4
9
16
>>> list( (x**2 for x in range(5)) )
[0, 1, 4, 9, 16]
>>>
```

Optimisation

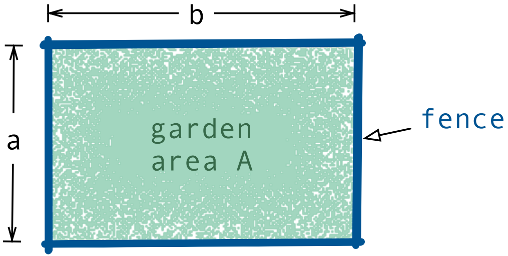
Optimisation example: garden fence



Optimisation problem:

- The shape of the fenced area must be a rectangle (side lengths a and b).
- We have $L = 100\,\mathrm{m}$ of fence available.
- We want to maximise the enclosed garden area $A = ab$.
- What are the optimal values for a and b?

Optimisation example: strategy

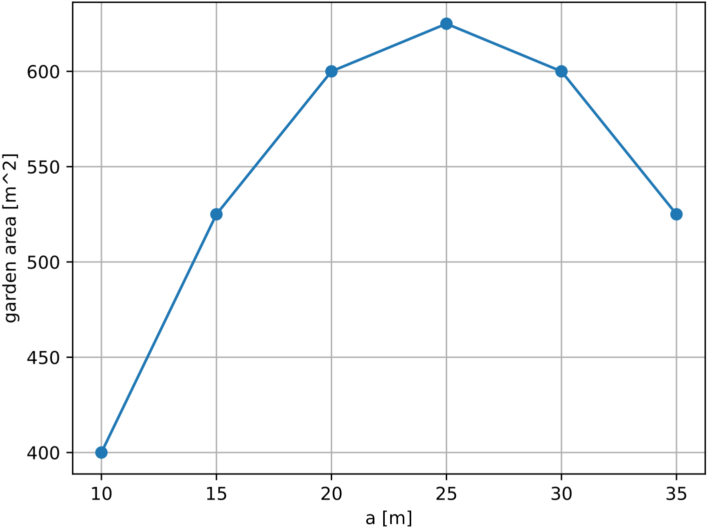


How do we find a and b that optimise the area $A(a, b)$?

- We know $L = 100\,\mathrm{m} = 2a + 2b$
- So we have only one unknown: when a is fixed, then b is given by $b = (L - 2a)/2$.
- Change a systematically to find best largest value of A.

```python
import matplotlib.pyplot as plt


def fenced_area(a):
    """Return area for garden with side length a.

    Given the side length a of a rectangular garden fence
    (with side lengths a and b), compute what side length
    b can be used for a total fence length of 100m.
    Return the associated area.
    """
    L = 100  # total length of fence in metre
    # for a given a, what is length b to use all 100m?
    #    L = 2*a + 2b   =>   b = (L - 2a) / 2
    b = (L - 2*a) / 2
```

```
# main program
side_lengths = []  # collect the side length a
areas = []  # collect the associated areas

# vary side length of fence a [in metres]
for a in range(10, 40, 5):
    side_lengths.append(a)
    areas.append(fenced_area(a))

plt.plot(side_lengths, areas, '-o')
plt.xlabel('a [m]')
plt.ylabel('garden area [m^2]')
plt.grid(True)
plt.savefig('optimisation-fence.pdf')
```


garden area [m^2]
600
550
500
450
400
10
15
20
25
30
35
a [m]

Optimisation example: "educational example"

We show one *strategy* to solve an optimisation problem with a simple example so we can focus on the strategy.

For the given fence problem:

- we can guess the correct answer
- there are better ways to find the result with the computer
- we can find the correct answer analytically

Analytical solution

- $A(a) = ab = a\frac{(L-2a)}{2} = \frac{aL}{2} - a^2$
- Find maximum using $\frac{\mathrm{d}A}{\mathrm{d}a} \stackrel{!}{=} 0 : \frac{\mathrm{d}A}{\mathrm{d}a} = \frac{L}{2} - 2a \Rightarrow a = \frac{L}{4}$
- $b = \frac{L-2a}{2} \Rightarrow b = \frac{L}{4}$
- Check $\frac{\mathrm{d}^2A}{\mathrm{d}a^2} = -2 < 0 \Rightarrow A\left(\frac{L}{4}\right)$ is maximum. ✓

```
commit fa7fe4b0c5c60fb97b941fb639cda6b7e29e45da
Author: Hans Fangohr <fangohr@users.noreply.github.com>
Date:   Sat Nov 2 15:53:22 2024 +0100

    complete lecture 5 preparation.
```