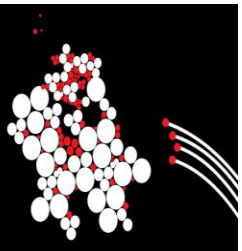
UNIVERSITY OF TWENTE.



Algorithmics

Pragmatics of design and realization of systems that compute for you

1

This lecture's agenda

Algorithm and workflow development: where to start and how to work

- Development = design + realization
- = strategic thinking + coding skills
- = first + second
- Ergo, you must have a plan.
- What if it fails? Debugging.

Documenting what you are deciding, what you are doing and how your code works

- For workflows, functions, classes
- Describe purpose, including expected inputs and outputs
- Describe computational strategy, possibly reference to lietrature
- Describe assumptions over the data
- Describe purpose of internal data structures

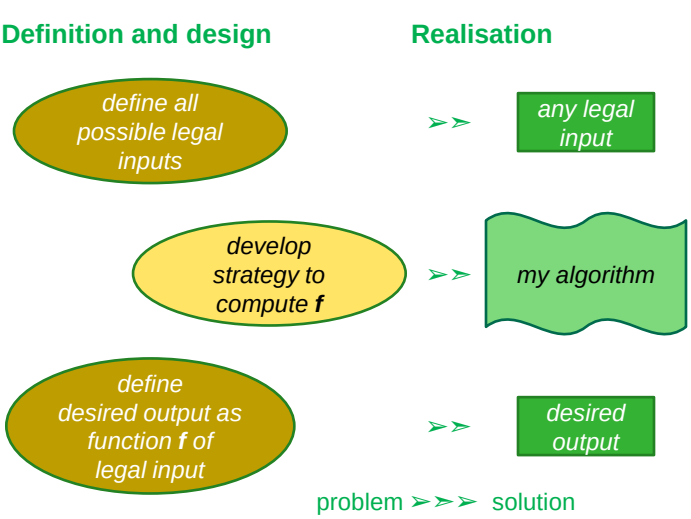
2

Algorithm and workflow development

- **Workflow**: a largish algorithm, usually comprising a number of different phases with rather distinct data processing functions. Where human interaction is involved, these may be different people.
- Top level structure of most workflows is a sequence of work phases.

- What we state below about algorithms and algorithm development applies to workflows also.

The algorithm abstractly



Some obvious solution structure

Leads to *natural structure* of every algorithm

any legal input

my algorithm

desired output

is given input legal?

determine the output

is the output the desired output?

5

So many problems … so many strategies

Know your problem

- Is the solution unique?
- Must I obtain all solutions?
- Is finding them hard?
- Does performance matter?
- Do I know similar problems (and how those have been solved)?

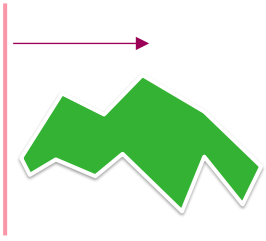
Know existing strategies

- Many algorithms are straightforward transformations from the input towards the output
- Identify embedded (smaller) problems and use solutions to these where they exist
- Mimick strategies applied elsewhere
- Recognize recurrence characteristics in your problem

6

Example strategies in geospatial data algorithms

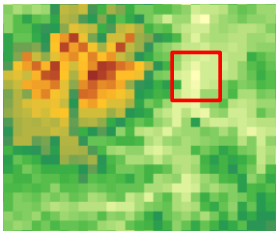
Working with **vector** data



Running a sweepline across the geometry

Going from one vertex on polygon boundary to the next

Working with **raster** data

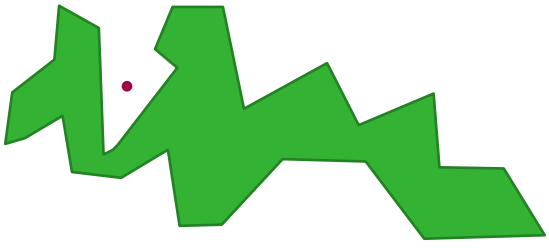


Passing a NxN kernel over an image (N preferably odd)

Visiting each raster cell except those at "edge" of raster

For example, point-in-polygon algorithm

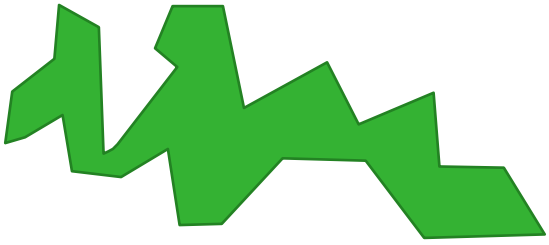
Classical problem: is a given point within a given polygon?



What is your strategy?

For example, polygon area size algorithm

Classical problem: what is the area size of a given polygon?



What is your strategy?

Third example: Assessing polygon boundary fits

Open problem: how well does one polygon data set match another?

What is your strategy?

Naming your coding constructs*

Principle: use meaningful, easy to interpret names that cannot lead to confusion or misinterpretation

- Context separates *users* of your code from *co-coders* of your code
- Apply conventions consistently
- Use English names and ASCII character set only
- The characters *l, 1, o, 0 O, I* are prone to problems as single character names
- Global and local variables, formal parameters, classes, functions, and methods are different animal types, so deserve different naming conventions.

** We discuss in Python context only. Similar rules apply to other environments.*

Python naming conventions (by PEP8)

Module & package: short , all lowercase, _ is allowed

Class: CapsWords convention, unless class is a callable and then function naming is appropriate

Type variable: as class names, short name preferred

Function & variable: lowercase, _ is allowed

Method: lowercase, starts with _ when method is non-public

Formal parameter (to function, method): lowercase, first par to instance method is "self" and to class method is "cls"

Constant: capitals, _ is allowed

Global variable: as function names, (make them rare)

Exception: is a class, so as class, preferably starting with prefix "Error"

Documentation*

Documentation is always done in English.

Docstrings

- Semi-required, more formal
- PEP257 suggestions
- The sw of *docutils* provides support

audience ≈ user

Comments

- Where deemed needed

audience ≈ coder

** We discuss in Python context only.*

Docstrings in detail

- A descriptive single string, associated with module, function, class, method or (complete) script.
- Becomes accessible *__doc__* attribute of these objects
- One-line or multiline triple quote mechanism:
  """ Return the polygon's area size in m2."""

- Reads as a command, and expresses the effect of the code.
- Ends with a period. No empty lines around a docstring.
- Does *not* repeat the function/method signature.
- Does explain the output/result:

"""Do XYZ, and return KLM."""

Multiline docstrings

- When more elaboration is justified.
- One-liner + empty line + elaboration
- Elaboration content depends on type of object (see below)

```
def determine_polygon_size(pol):
    """Return the polygon's area size in m2.

    Arguments:
    pol -- shapely Polygon object with srid=4326
    """
```


Script multiline docstrings

Script: stand-alone program

- Provides insight in the script's function and usage syntax, any dependency on operating system environment variables, and (non-)existence of files.
- Can be quite elaborate (10s of lines), depending on complexity of use, and options that the script provides.
- Sufficient for a novice user, but also documenting all options for advanced use.

Function & method multiline docstrings

Function: stand-alone component in a script
Method: function associated with (defined within) a class

- Provides insight in the function and usage syntax, return value(s), side effects (if any), possible exceptions raised, and the preconditions of use.
- Which optional arguments are available, and what do they represent?
- Which keyword arguments are available, and what do those represent?

Class & module multiline docstrings

Class: a user-defined object type, as we will discuss
Module: a package of classes, functions etc. in a single source file

- See the PEP257 for conventions of docstrings on classes and modules

Comments

- Co-developed with the code: train of thought & decisions
  - Thus: change of code means change of comment
- Explains how the code operates, also 6 months after coding
- Must never confuse the reader/coder
- Starts at #, followed by a space and continues after

Two types:

- **Block comments**
- **Inline comments**

Block comments

- Serve to *describe approach* of upcoming code segment, indented as much as that code.
- Only comment, no code included

```
# Block comments provide well-written, complete sentences.  They
# typically explain the approach that the code block below takes
# to compute something, and may also explain the data structures
# that are used to that end.
#
# Block comments can have multiple paragraphs.  These are
# indicated by an "empty" line between them.  Observe the double
# spaces after each complete line.  Never capitalize any of the
# variable/function/parameter names in your comments.
```

Inline comments

- Use *sparingly*, behind and on same line as the code.
- Explain only what needs explanation.
- Two spaces, then #, and then another space before comment starts

```
tck, u = splprep(pts.T, u=None, s=0.0, per=1)  # Use points array as transposed
u_new = np.linspace(u.min(), u.max(), 10000)
                                               # Use 10k here, should become N per km
x_new, y_new = splev(u_new, tck, der=0)

plt.plot(x_new, y_new, 'b--')                  # Use blue dashed lines
plt.show()
```