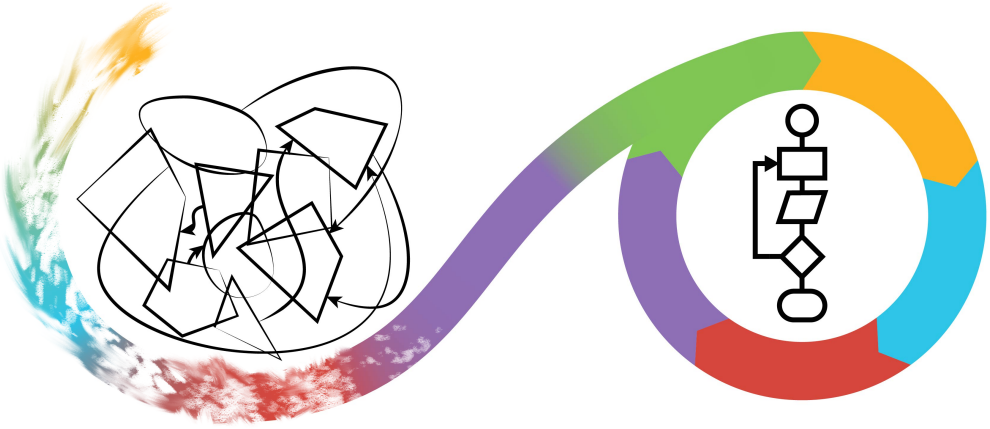
Data structures

Richèl Bilderbeek

https://github.com/UPPMAX/programming_formalisms/blob/main/data_structures/data_structures_lecture/data_structures_lecture.qmd



Problem

Are there classes that can help me solve problems more elegantly?

How do I write these myself?

Python classes

- (`list`: heterogeneous container)
- (`tuple`: immutable `list`)
- `set`: sets
- `dict`: dictionary
- Regular expressions: text patterns

From Python 'Data Structures' documentation

`set`

Sorted collection of unique elements.

```
data = [3, 1, 4, 1, 5]
s = set(data)
assert 3 in s
assert list(s) == [1, 3, 4, 5]
```

- No need to check for elements existing twice

`set` example for `are_primes`

- `are_primes` determines of each value in a list if it is prime yes/no
- Problem: values in that list can occur multiple times, e.g. `[42, 42, 42]`
- Solution with a `set`:
  - Collect all unique values in the input
  - Put the unique values that are prime in a set
  - Check each input value to be in the set of primes

`dict`

A dictionary:

```
periodic_table = dict({1: "Hydrogen", 2: "Helium", 3: "Lithium"})
periodic_table[2] = "helium"
assert periodic_table[2] == "helium"
```

- Commonly uses as a look-up table
- A look-up table can store the results of earlier calculations

`dict` example for `are_primes`

- `are_primes` determines of each value in a list if it is prime yes/no
- Problem: values in that list can occur multiple times, e.g. `[42, 42, 42]`
- Solution with a `set`:
  - Collect all unique values in the input
  - Store for each unique values if it is prime yes/no
  - Look up each input value in the prime look-table

Regular expressions

A state-machine for a pattern in text

```
import re
dna_regex = re.compile("^[ACGT]*$")
assert dna_regex.match("")
assert dna_regex.match("A")
assert dna_regex.match("CA")
assert dna_regex.match("GCA")
assert dna_regex.match("TGCA")
assert dna_regex.match("TGCAAAAAA")
assert not dna_regex.match("nonsense")
```

- https://docs.python.org/3/library/re.html

Writing our own classes

Q: What is a good class?

. . .

A:

- guarantees its stored data is valid, e.g the class `DnaSequence` is probably a string of one or more A, C, G and T
- the quality requirements for a function, among others a good interface
- writing a design, documentation and tests all help

Class anatomy

- `__init__`: instantiation operation (a.k.a. 'constructor')
- Private variables: what it secretly is
- Methods: how to work on it

```
class DnaSequence:
    def __init__(self, sequence):
        assert is_dna_string(sequence)
        self._sequence = sequence # convention

    def get_str(self):
        return self._sequence

a = DnaSequence("ACGT")
assert a.get_str() == "ACGT"
```

Private variables are a social convention

Use of `_` before the name of a private variable is a social convention!

```
self._sequence = sequence # convention
```

Nothing stops you from:

```
a._sequence = "XXX"
assert a.get_str() == "XXX"
```

Some other programming languages offer stronger guarantees.

Inheritance and polymorphism

> C++ is a horrible language. It's made more horrible by the fact that a lot of substandard programmers use it, to the point where it's much much easier to generate total and utter crap with it.
>
> Linus Torvalds, 2007-09-06

Inheritance and polymorphism

- Can create class hierarchies
  - 'All Animal objects can make a sounds'
- Easy to abuse, hard to use correctly
- Design Patterns are known to work well

Design Patterns

- Named proven-to-work class hierarchies
- Examples:
  - Singleton: make sure an object exists once
  - Strategy: multiple ways to the same thing
  - Memento: allow for undo functionality
- Classic: the Gang of Four book

Class design

- Python classes
- C++ Core Guidelines

Recap