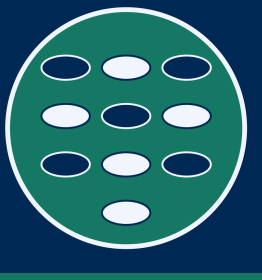
Norges miljø- og biovitenskapelige universitet

Institutt for datavitenskap



Digitalisering på Ås

DAT121
Introduction to data science

2 Data and objects

2.1 Object-oriented programming in Python
2.2 Inheritance and class hierarchies
2.3 Conceptual modelling

Norwegian University of Life Sciences

Schedule for 17th and 18th August

Thursday, 17th August 2023

09.15 – 10.00 discussion and Q&A
1.5 Python libraries

10.15 – 11.00 problem solving and start of first lecture on data and objects
2.1 OOP in Python

11.15 – 12.00 problem solving and continued first lecture on data and objects
2.2 Inheritance
2.3 Conceptual modelling

13.15 – 15.00 tutorial session

Friday, 18th August 2023

09.15 – 10.00 discussion and Q&A

10.15 – 11.00 second lecture on data and objects
2.4 Semantic interoperability
2.5 Knowledge graphs
2.6 Querying

11.15 – 12.00 Fadi Al Machot's presentation on research and Master topics

The afternoon of 18th August is reserved for the immatriculation.

Programming paradigms

Imperative programming
- It is stated, instruction by instruction, what the processor should do
- Control flow implemented by jumps (**goto**)

Structured programming
- Same, but with **higher-level control flow**
- Contains "instruction by instruction" code

Procedural programming
- **Functions** (procedures) as **highest-level structural unit** of code
- Still contains loops, *etc.*, for control flow within a function

Object-oriented programming (OOP)
- **Classes** as **highest-level structural unit** of code; objects instantiate classes
- Still contains functions, *e.g.*, as methods

Programming paradigms based on **describing the solution** rather than computational steps:

Functional programming
(also: "declarative programming")

Constraint programming

Logic programming

Generic programming
(introduces ideas from declarative and logical methods into OOP)

Recursion in procedural programming

Recursion is the process of defining the solution to a problem (or the solution to a problem) in terms of a simpler or smaller instance of the same problem.

The base case (or a base case) is reached when the problem has been simplified to the utmost

Image from: https://www.therussianstore.com/blog/the-history-of-nesting-dolls

Recursion is a form of decomposition:

solution(k) ≡ recursive_step(solution(< k))

solution(⊥) ≡ base_case_solution

Recursion in procedural programming

Recursion is the process of defining the solution to a problem (or the solution to a problem) in terms of a simpler or smaller instance of the same problem.

The base case (or a base case) is reached when the problem has been simplified to the utmost

Multiple recursion decomposes a problem into more than one simplified instance

Image from: https://www.therussianstore.com/blog/the-history-of-nesting-dolls

Recursion is a form of decomposition:

solution(k) ≡ recursive_step(solution$_1$($< k$), solution$_2$($< k$), …)

solution(⊥) ≡ base_case_solution

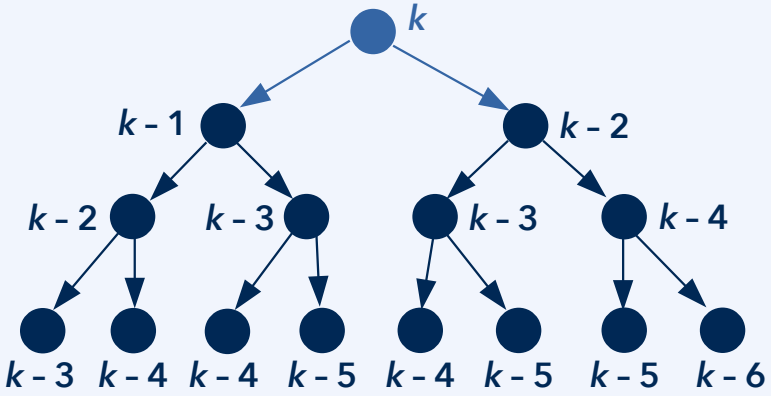
Multiple recursion

The **Fibonacci numbers** constitute a mathematical sequence that is defined by multiple recursion:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_k = F_{k-1} + F_{k-2}, \text{ for } k > 1$$

0, 1, 1, 2, 3, 5, 8, 13, …

While the definition is most conveniently given in the form of a **recursion**, the numerical implementation would usually proceed by **iteration**. Compare the code employing a loop with that obtained by a direct calque of the definition.



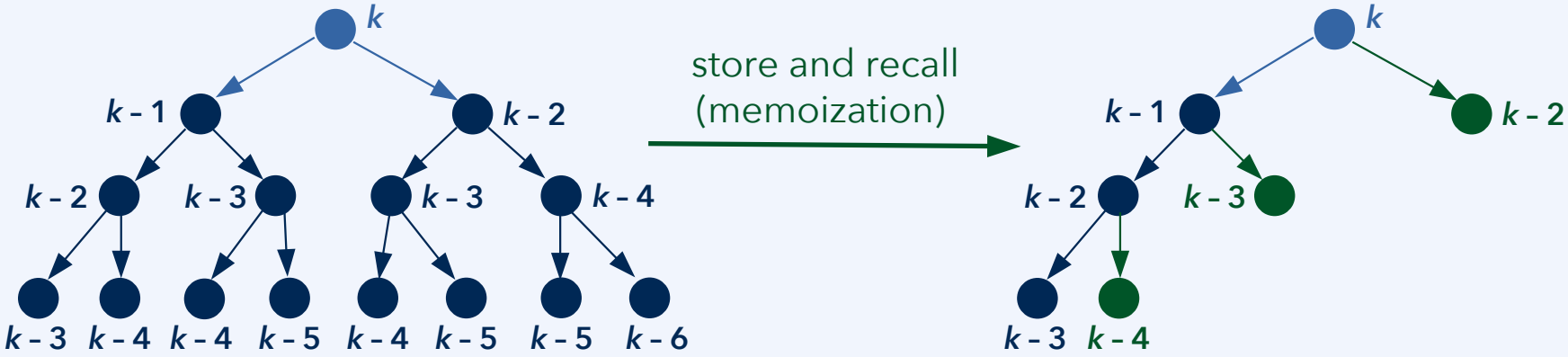
Multiple recursion: Dynamic programming

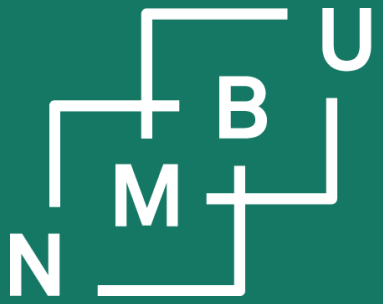
The **Fibonacci numbers** constitute a mathematical sequence that is defined by multiple recursion:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_k = F_{k-1} + F_{k-2}, \text{ for } k > 1$$

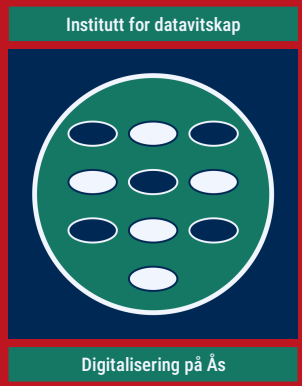
0, 1, 1, 2, 3, 5, 8, 13, …

While the definition is most conveniently given in the form of a **recursion**, the numerical implementation would usually proceed by **iteration**. Compare the code employing a loop with that obtained by a direct calque of the definition.





Noregs miljø- og biovitskaplege universitet



2 Python basics

2.1 Object-oriented Python

DAT121

17. august 2023

Programming paradigms

Imperative programming

- It is stated, instruction by instruction, what the processor should do
- Control flow implemented by jumps (**goto**)

Structured programming

- Same, but with **higher-level control flow**
- Contains "instruction by instruction" code

Procedural programming

- **Functions** (procedures) as **highest-level structural unit** of code
- Still contains loops, *etc.*, for control flow within a function

Object-oriented programming (OOP)

- **Classes** as **highest-level structural unit** of code; objects instantiate classes
- Still contains functions, *e.g.*, as methods

Programming paradigms based on **describing the solution** rather than computational steps:

Functional programming
(also: "declarative programming")

Constraint programming

Logic programming

Generic programming
(introduces ideas from declarative and logical methods into OOP)

Why object orientation?

The job of variables is to store data. In object oriented programming (OOP) the focus is on *how data belong together* and how we can facilitate *safe and correct access to data*. How do data-centered tools (DBs, *etc.*) present data?

Example: "Largest cities by country" query on Wikidata.

city	cityLabel	population	country	countryLabel	loc
wd:Q172	Toronto	2731571	wd:Q16	Canada	Point(-79.386666666 43.670277777)
wd:Q1490	Tokyo	14047594	wd:Q17	Japan	Point(139.691722222 35.689555555)
wd:Q585	Oslo	693494	wd:Q20	Norway	Point(10.738888888 59.913333333)
wd:Q1761	Dublin	553165	wd:Q27	Republic of Ireland	Point(-6.260277777 53.349722222)
wd:Q1781	Budapest	1723836	wd:Q28	Hungary	Point(19.040833333 47.498333333)
wd:Q2807	Madrid	3305408	wd:Q29	Spain	Point(-3.7025 40.416666666)
wd:Q60	New York City	8804190	wd:Q30	United States of America	Point(-74.0 40.7)
wd:Q240	Brussels-Capital Region	1218255	wd:Q31	Belgium	Point(4.3525 50.846666666)
wd:Q1842	Luxembourg	128512	wd:Q32	Luxembourg	Point(6.132777777 49.610555555)
wd:Q1757	Helsinki	643272	wd:Q33	Finland	Point(24.93417 60.17556)
wd:Q1754	Stockholm	978770	wd:Q34	Sweden	Point(18.068611111 59.329444444)
wd:Q1748	Copenhagen	644431	wd:Q35	Denmark	Point(12.568888888 55.676111111)
wd:Q270	Warsaw	1790658	wd:Q36	Poland	Point(21.011111111 52.23)

object-oriented programming

Why object orientation?

The job of variables is to store data. In object oriented programming (OOP) the focus is on *how data belong together* and how we can facilitate *safe and correct access to data*. How do data-centered tools (DBs, *etc.*) present data?

Example: "Largest cities by country" query on Wikidata.

city	cityLabel	population	country	countryLabel	loc
wd:Q172	Toronto	2731571	wd:Q16	Canada	Point(-79.386666666 43.670277777)
wd:Q1490	Tokyo	14047594	wd:Q17	Japan	Point(139.691722222 35.689555555)
wd:Q585	Oslo	693494	wd:Q20	Norway	Point(10.738888888 59.913333333)
wd:Q1761	Dublin	553165	wd:Q27	Republic of Ireland	Point(-6.260277777 53.349722222)
wd:Q1781	Budapest	1723836	wd:Q28	Hungary	Point(19.040833333 47.498333333)
wd:Q2807	Madrid	3305408	wd:Q29	Spain	Point(-3.7025 40.416666666)
wd:Q60	New York City	8804190	wd:Q30	United States of America	Point(-74.0 40.7)
wd:Q240	Brussels-Capital Region	1218255	wd:Q31	Belgium	Point(4.3525 50.846666666)
wd:Q1842	Luxembourg	128512	wd:Q32	Luxembourg	Point(6.132777777 49.610555555)
wd:Q1757	Helsinki	643272	wd:Q33	Finland	Point(24.93417 60.17556)
wd:Q1754	Stockholm	978770	wd:Q34	Sweden	Point(18.068611111 59.329444444)
wd:Q1748	Copenhagen	644431	wd:Q35	Denmark	Point(12.568888888 55.676111111)
wd:Q270	Warsaw	1790658	wd:Q36	Poland	Point(21.011111111 52.23)

Class definition (Python syntax)

See the examples from the Python tutorial, Section 9.3.5:[1]

```
class Dog:
    kind = 'canine'         # class variable shared by all instances
    def __init__(self, name):
        self.name = name    # instance variable unique to each instance
>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # shared by all dogs
'canine'
>>> e.kind                  # shared by all dogs
'canine'
>>> d.name                  # unique to d
'Fido'
>>> e.name                  # unique to e
'Buddy'
```

We also call this an **attribute** or a **field** of the class

```
class Dog:
    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog
    def add_trick(self, trick):
        self.tricks.append(trick)
>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

[1]https://docs.python.org/3/tutorial/classes.html#class-and-instance-variables

Class definition (Python syntax)

jupyter book-index (autosaved)

File Edit View Insert Cell Kernel Widgets Help Not Trusted Python 3

Run Code

In [1]:

```
class BookIndex:
    def __init__(self):
        self._chapter = 1
        self._section = 1
        self._page = 1

    def next_chapter(self):
        self._chapter += 1
        self._section = 1
        self._page += 1
        return self._chapter

    def next_section(self):
        self._section += 1
        return self._section

    def next_page(self):
        self._page += 1
        return self._page

    def out(self):
        print("Section ", self._chapter, \
              ".", self._section, ", p. ", \
              self._page, sep="", end="\n")
```

What are the attributes of the class BookIndex?

What is the meaning of the keyword "self"?

book-index.ipynb

In [4]:

```
idx = BookIndex()
idx._chapter = 1
idx._section = 8
idx._page = 8

idx.out()
```

```
Section 1.8, p. 8
```

Class definition (Python syntax)

jupyter book-index (autosaved)

File Edit View Insert Cell Kernel Widgets Help Not Trusted Python 3

Run Code

In [1]:

```
class BookIndex:
    def __init__(self):
        self._chapter = 1
        self._section = 1
        self._page = 1

    def next_chapter(self):
        self._chapter += 1
        self._section = 1
        self._page += 1
        return self._chapter

    def next_section(self):
        self._section += 1
        return self._section

    def next_page(self):
        self._page += 1
        return self._page

    def out(self):
        print("Section ", self._chapter, \
              ".", self._section, ", p. ", \
              self._page, sep="", end="\n")
```

In [4]:

```
idx = BookIndex()
idx._chapter = 1
idx._section = 8
idx._page = 8

idx.out()
```

```
Section 1.8, p. 8
```

Python tutorial, Section 9.6:

"**private**" instance variables that cannot be accessed except from inside an object don't exist in Python.

However, there is a **convention** that is followed by most Python code: a name **prefixed with an underscore** (e.g. _spam) should be treated as a **non-public** part of the API (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice.

Why is it **bad practice** to do this?
What should we do instead?

Python classes compared to C/C++ structures

```
In [1]: class BookIndex:
            def __init__(self):
                self._chapter = 1
                self._section = 1
                self._page = 1

            def next_chapter(self):
                self._chapter += 1
                self._section = 1
                self._page += 1
                return self._chapter

            def next_section(self):
                self._section += 1
                return self._section

            def next_page(self):
                self._page += 1
                return self._page

            def out(self):
                print("Section ", self._chapter, \
                      ".", self._section, ", p. ", \
                      self._page, sep="", end="\n")

In [2]: idx = BookIndex()
        idx._chapter = 1
        idx._section = 8
        idx._page = 25

        idx.out()

        Section 1.8, p. 25

In [ ]: def start_chapter(b):
            b.next_chapter()
            b.out()

In [ ]: start_chapter(idx)
        idx.out()
```

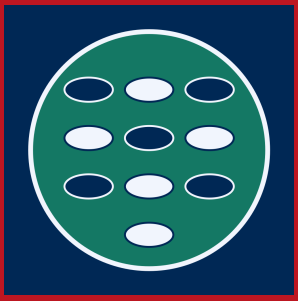
```
struct BookIndex
{
  int chapter = 1;
  int section = 1;
  int page = 1;

  int next_chapter();

  int next_section();

  int next_page();

  void out() const;
}
```

```
int BookIndex::next_chapter( ) {
  this->chapter++;
  this->section = 1;
  this->page++;
  return this->chapter;
}

int BookIndex::next_section( ) {
  this->section++;
  return this->section;
}

int BookIndex::next_page( ) {
  this->page++;
  return this->page;
}

void BookIndex::out( ) const {
  cout << "Section " << this->chapter
       << "." << this->section
       << ", p. " << this->page << "\n";
}
```

How do Python, Java, and C++ deal with argument passing?

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap



Digitalisering på Ås

2 Data and objects

2.1 Object-oriented Python

2.2 Inheritance and taxonomy

DAT121

17. august 2023

Inheritance (Python syntax)

See Chapter 9 of the Python tutorial as follows:

9.5. Inheritance

Of course, a language feature would not be worthy of the name “class” without supporting inheritance. The syntax for a derived class definition looks like this:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

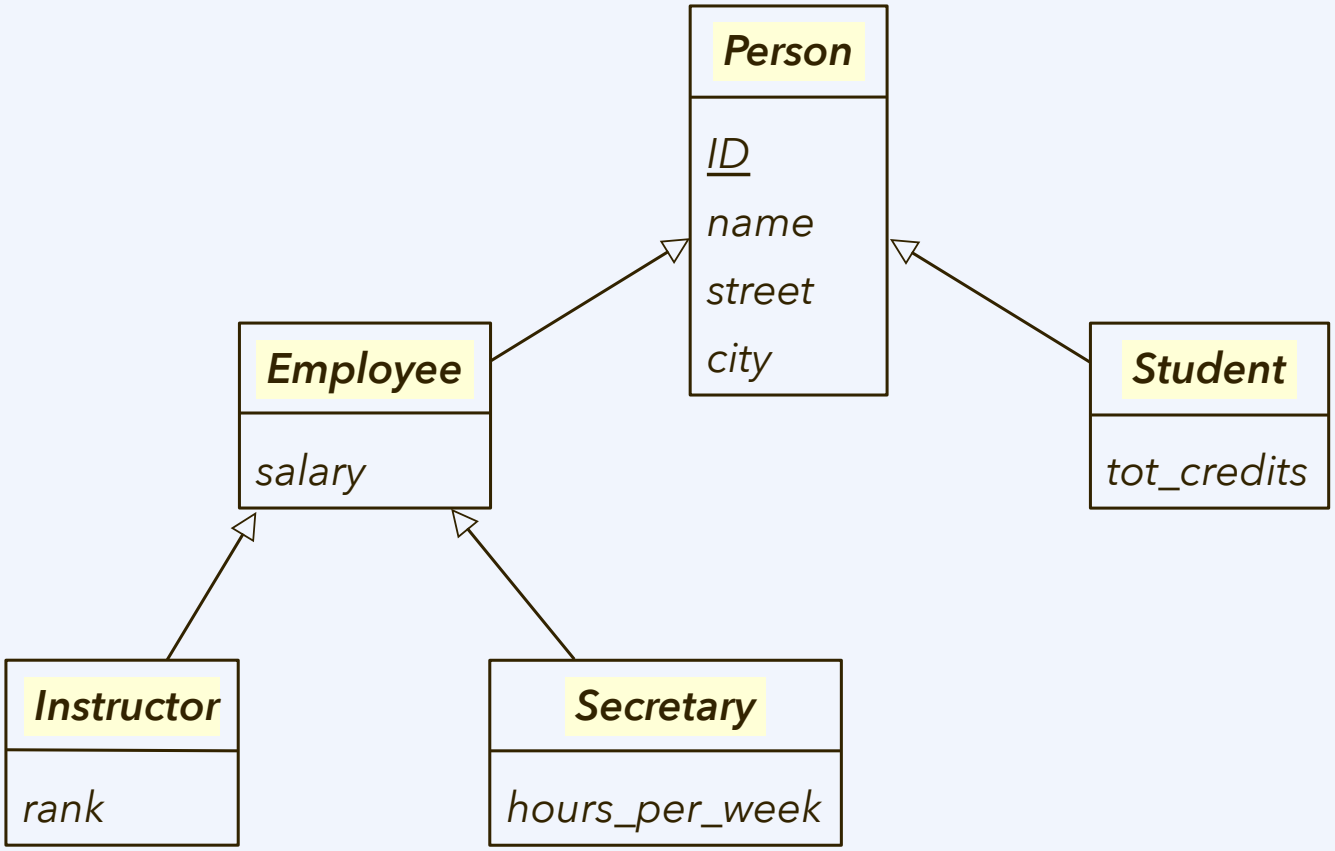
9.5.1. Multiple Inheritance

Python supports a form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Conceptual hierarchy or taxonomy

Consider the following example from Silberschatz *et al.* (Fig. 6.18):



Conceptual hierarchy or taxonomy

In Python programming, complicated class hierarchies are possible, but rarely used. This includes **multiple inheritance** (diamond structure in the taxonomy[1]).

In Java, but also C++, it is common to find deep class hierarchies, *e.g.*:

javax.security.auth.Destroyable
    java.security.PrivateKey (also extends java.security.Key)
java.security.interfaces.RSAKey
    java.security.interfaces.RSAPrivateKey (also extends java.security.PrivateKey)
        java.security.interfaces.RSAMultiPrimePrivateCrtKey
        java.security.interfaces.RSAPrivateCrtKey
    java.security.interfaces.RSAPublicKey (also extends java.security.PublicKey)
java.io.Serializable
    java.security.Key
        java.security.PrivateKey (also extends javax.security.auth.Destroyable)
        java.security.PublicKey

[1]Python tutorial, Sec. 9.5.1: "all cases of multiple inheritance exhibit one or more diamond relationships (where at least one of the parent classes can be accessed through multiple paths from the bottommost class). For example, all classes inherit from object, so any case of multiple inheritance provides more than one path to reach object."

Related teaching activities: INF205

INF205: Resource-efficient programming (spring term – used to be in autumn)

This course introduces students with experience in high-level programming languages (*e.g.*, Python) to programming in a compiled programming language with explicit **memory management**, with a focus on **efficient use of computational resources**.

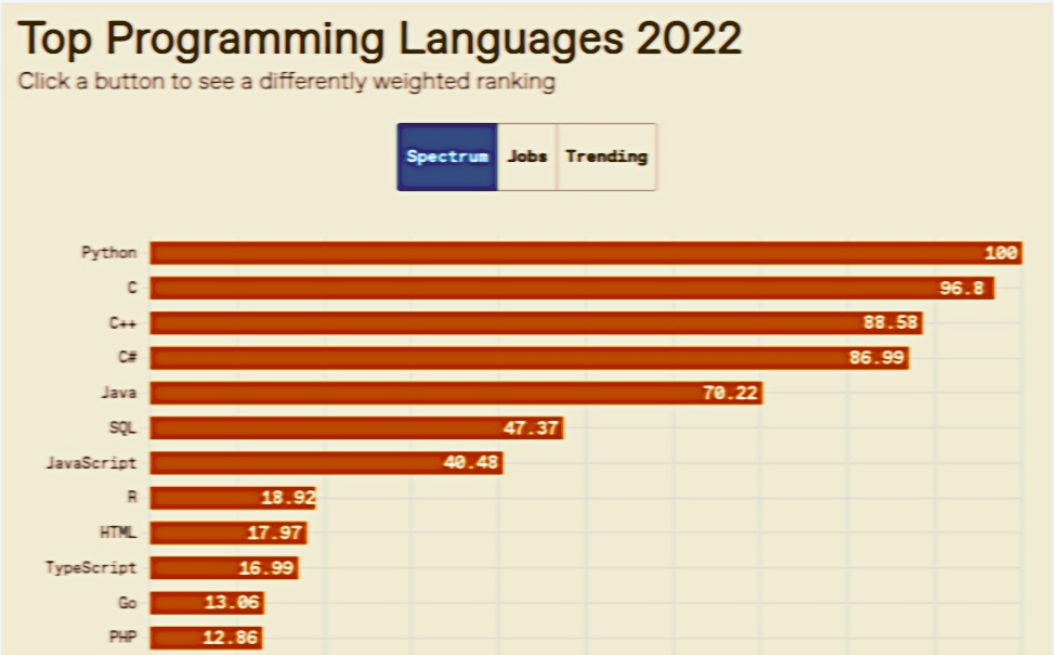
Specific topics are:

- Modern C++ syntax and semantics
- Compiling and building projects
- Pointers, memory allocation and deallocation
- Working with the C++ Standard Library
- Generic programming with templates
- Implementing containers from first principles
- Interprocess communication (MPI)
- Programming and sustainability
- Responsible use of high-performance computing infrastructure
- Interfacing with ROS (*e.g.*, for embedded systems)

Related teaching activities: INF205

INF205: Resource-efficient programming (spring term – used to be in autumn)

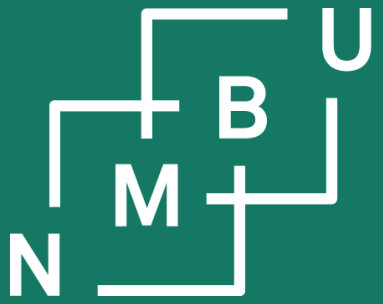
This course introduces students with experience in high-level programming languages (*e.g.*, Python) to programming in a compiled programming language with explicit **memory management**, with a focus on **efficient use of computational resources**.



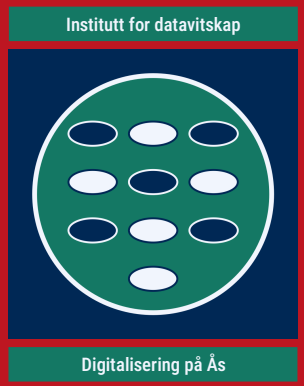
Python may be nice and elegant, but it is often inefficent. We still see C, C++, and C# on the ranks 2 to 4.

The INF205 module is the only one in data science teaching non-high-level programming, with control over memory.

S. Cass, *IEEE Spectrum*, https://spectrum.ieee.org/top-programming-languages-2022, **2022**.



Noregs miljø- og biovitskaplege universitet



2 Data and objects

2.1 Object-oriented Python

2.2 Inheritance and taxonomy

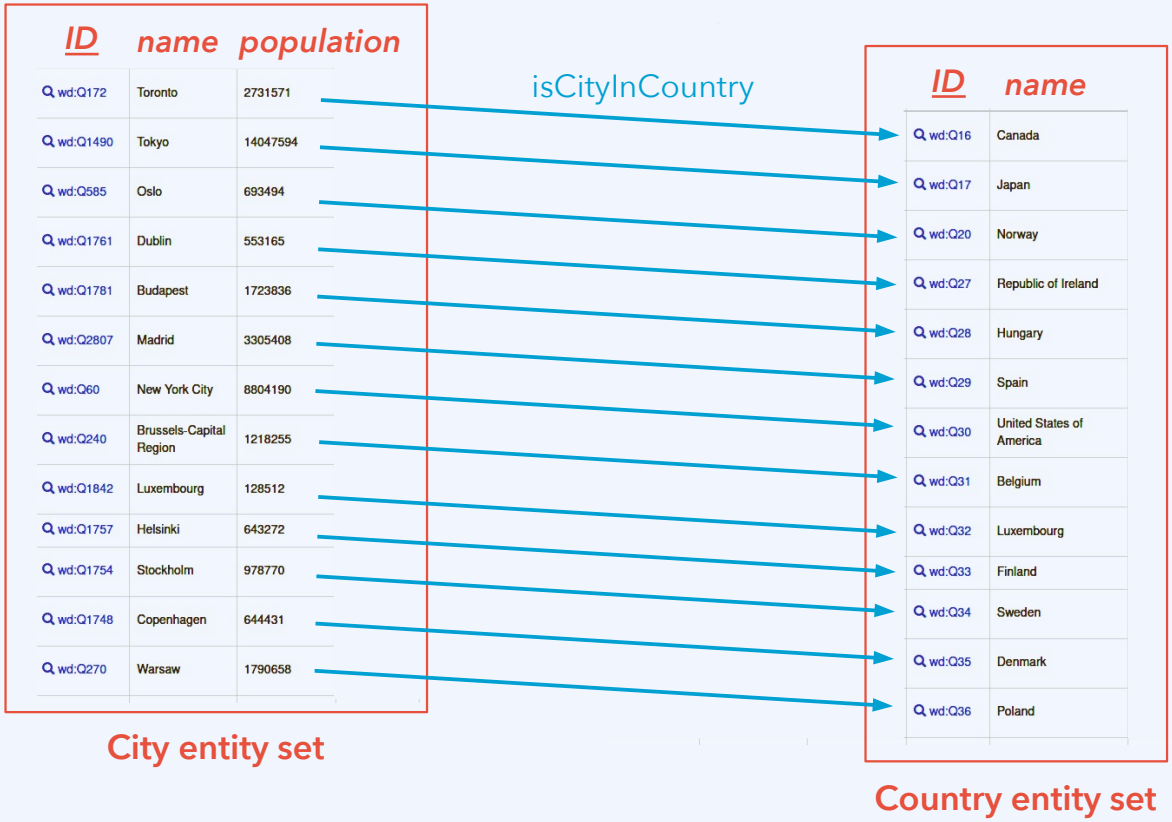
2.3 Conceptual modelling

DAT121

17. august 2023

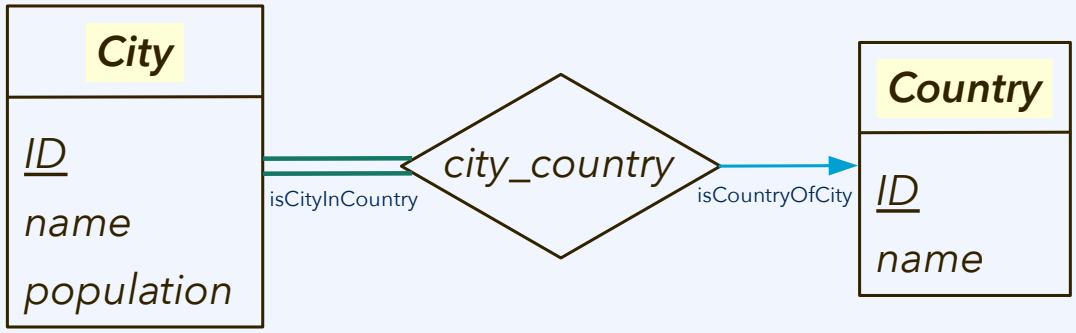
Entity-relationship (E-R) diagrams

particular: individual relationship **property**



Entity-relationship (E-R) diagrams

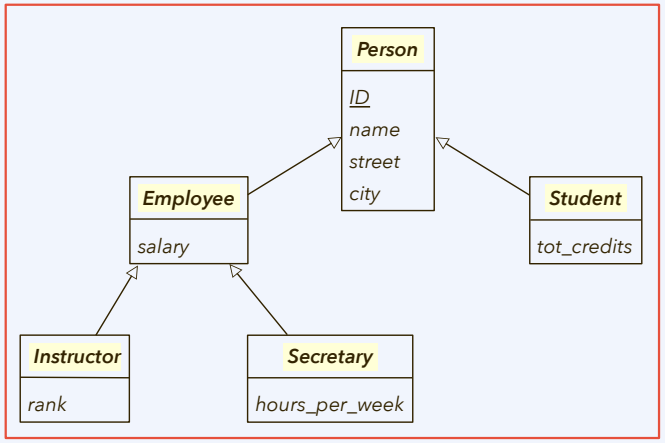
particular:	**individual**	**relationship**	**property**
universal:	concept	relation	**attribute**



"every City is in such a relationship"

"it is an *N*-to-1 relation from Cities to Countries"

This was also an entity-relationship diagram:

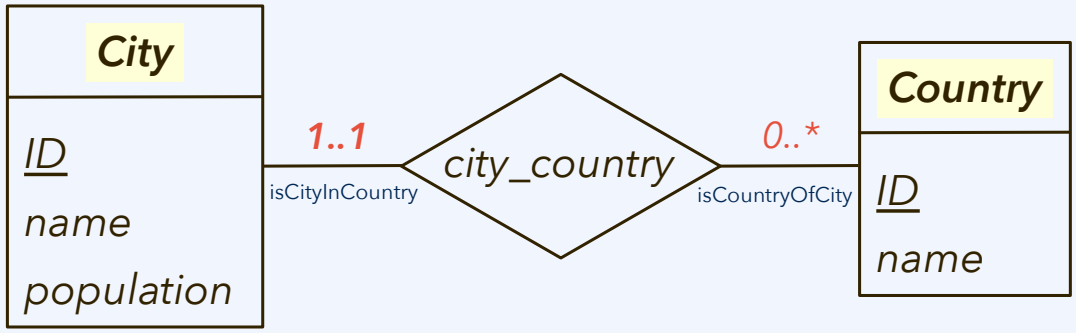


More on entity-relationship diagrams:

- Silberschatz *et al.*, *Database System Concepts*, Chapter 6
- https://en.wikipedia.org/wiki/Entity-relationship_model

Entity-relationship (E-R) diagrams

particular:	**individual** (entity, object)	**relationship**	**property** (sometimes: attribute)
universal:	**concept** (entity type, class)	**relation** (relationship type) (in OWL: ObjectProperty)	**attribute** (sometimes: attribute type) (in OWL: DatatypeProperty)

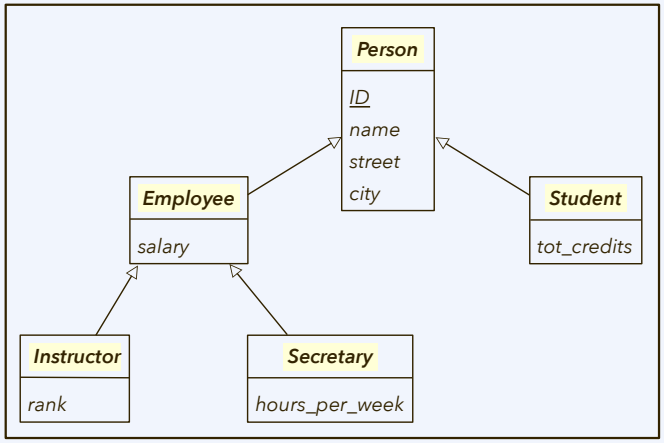


using cardinality constraints:

"each City is in exactly one Country"

"but there can be arbitrarily many (from 0 to infinity) Cities in each Country"

This was also an entity-relationship diagram:



concept relation

Particulars vs. universals

particular:	**individual** (entity, object)	**relationship**	**property**
universal:	**concept** (entity type, class)	**relation** (relationship type)	**attribute**

Definition: A **concept** is *a universal that is only instantiated by individuals*.

- From SKOS, a semantic artefact for organizing conceptual schemes: *"Concepts are the units of thought - ideas, meanings, or (categories of) objects and events – which underlie many knowledge organization systems"* (Isaac & Summers 2009).
- In many settings, including in object-oriented programming, a concept is usually called a **class**. In E-R diagram terminology, it is called an **entity type**.
- E-R terminology distinguishes between an entity type and the corresponding **entity set**, *i.e.*, the set of all individuals that instantiate the entity type. In nominalist ontology, these two are the same – a universal is the set of its individual instances.

concept relation

More about SKOS:
https://www.w3.org/TR/skos-primer/

Generic programming

Relying on clearly characterized concepts in object orientation is also called **generic programming** (GP), which can be seen as its own programming paradigm, building on OOP, but going beyond it; "by implementing programs generically, a single implementation can be used for many different types".[1]

Modern C++ supports such design by (1) inheritance and (2) templates.

```
template<typename SeqnT, ...>
  void test_sequence(SeqnT* sqn, ...)
{ ... }
```

[1]L. Escot, J. Cockx, Proc. *ACM Prog. Lang.* **6**: 625–649, doi:10.1145/3547644, **2022**.

Generic programming

Relying on clearly characterized concepts in object orientation is also called **generic programming** (GP), which can be seen as its own programming paradigm, building on OOP, but going beyond it; "by implementing programs generically, a single implementation can be used for many different types".[1]

Modern C++ supports such design by (1) inheritance and (2) templates.

From C++20 onward, **concepts** are introduced as GP language constructs. They describe requirements for a type (*e.g.*, it must provide an operator such as "<<", a particular method, or we must be able to add it to an integer, …).

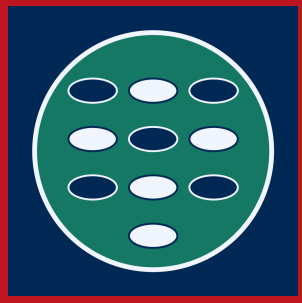
```
// old style: does not make clear what
// we expect from the class SeqnT
template<typename SeqnT, …>
  void test_sequence(SeqnT* sqn, …)
{ … }
```

```
// new style, where we would define Sequence
// as a concept (and not as an abstract class)
template<Sequence SeqnT, …>
  void test_sequence(SeqnT* sqn, …)
{ … }
```

[1] L. Escot, J. Cockx, Proc. *ACM Prog. Lang.* **6**: 625–649, doi:10.1145/3547644, **2022**.

Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap



Digitalisering på Ås

Conclusion

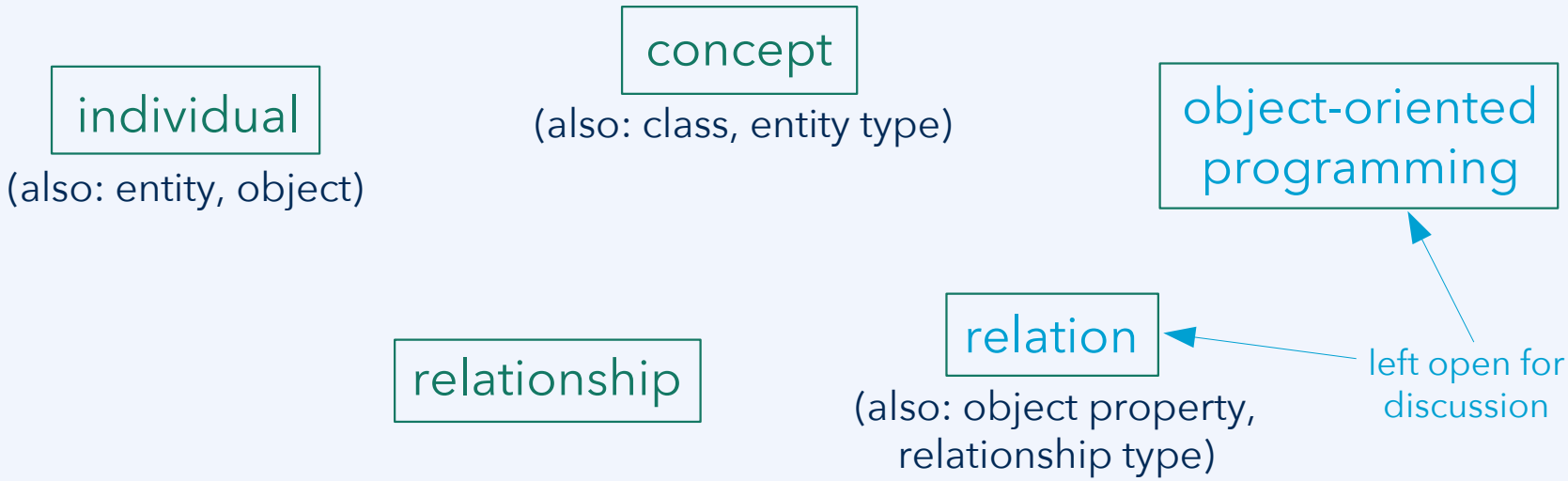
DAT121

17. august 2023

Glossary terms

Proposed glossary[1] terms:

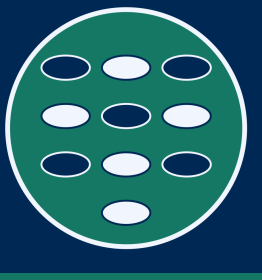
- How do we best define them? Is the definition controversial?
- What is the best translation into Norwegian bokmål/nynorsk?
- Are there more key concepts that would require an agreed definition?



[1]https://home.bawue.de/~horsch/teaching/dat121/glossary-en.html

Norges miljø- og biovitenskapelige universitet

Institutt for datavitenskap



Digitalisering på Ås

DAT121
Introduction to data science

2 Data and objects

2.1 Object-oriented programming in Python
2.2 Inheritance and class hierarchies
2.3 Conceptual modelling