Excursion - History of programming

Programs as sequences of statements

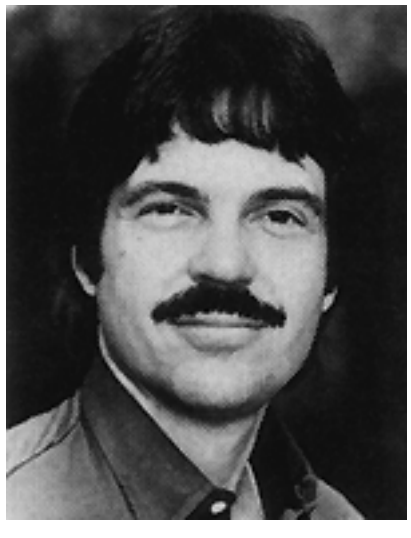
When Margaret Hamilton programmed the flight software for NASA's Apollo program, which landed humans on the moon for the first time in the 1960's she wrote programs in a way similar to your programs in the first four sessions of the seminar.

Structured programming - Procedural programs

When we in session five introduced functions and modules to structure your programs, we jumped to the late 60s/70s where people like Niklaus Wirth published languages like Algol W, Pascal, Modula, etc., which popularized the idea of reusable code in functions (procedures) and modules (units).

Object-oriented programming

Also in the 70s people like Alan Kay were arguing that it would make more sense to organize code according to shared data (properties) and (behavior).



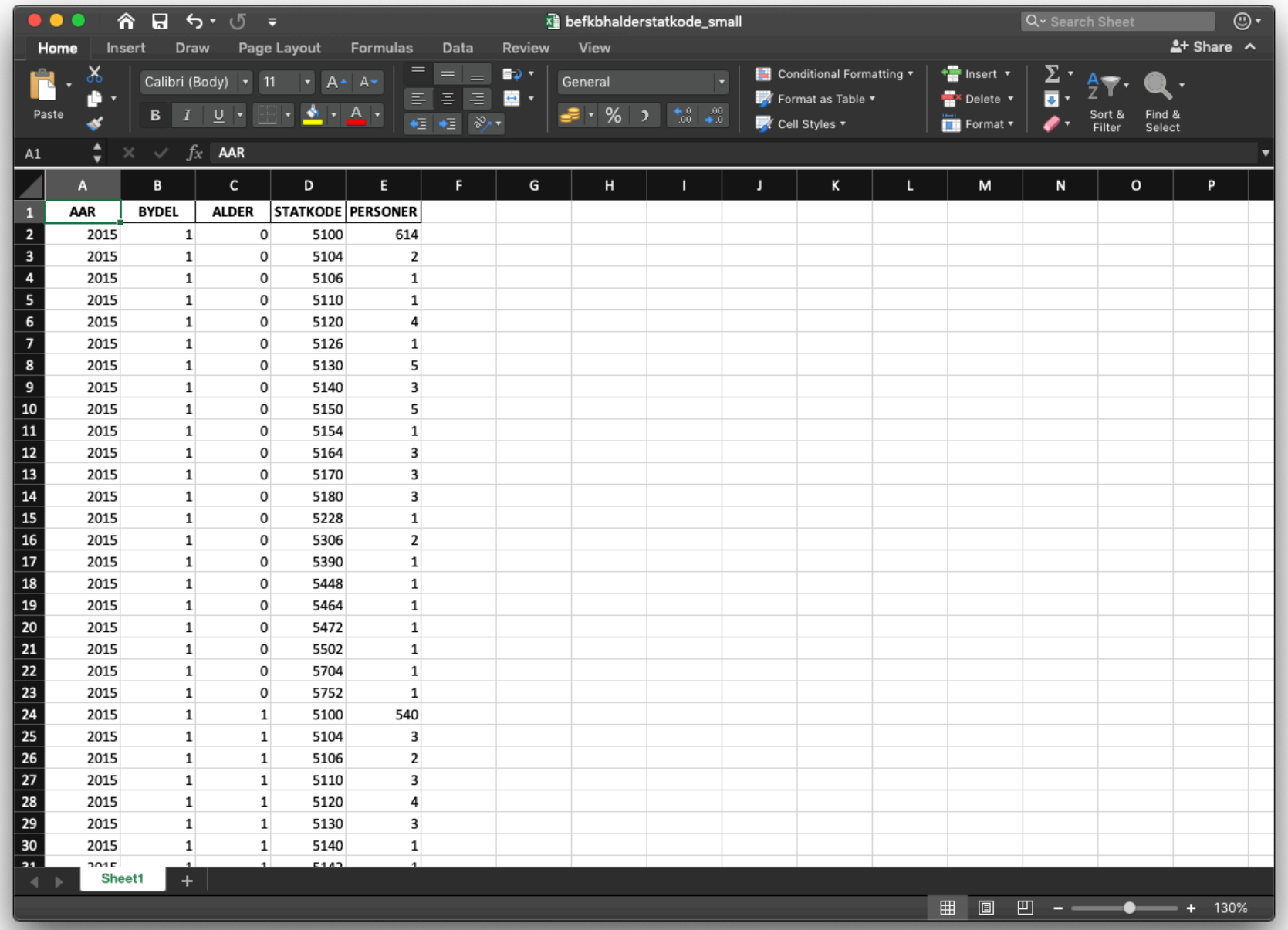
This is where we are now, after Friday's lecture. Especially in Denmark it is the predominant programming paradigm. Likely, almost all software that you will meet in your professional lives is written according to this paradigm.

Applying the object-oriented paradigm

- We saw how to work with cats, foxes and turtles as classes
- But why does object-orientation matter in practice?

You know this, don't you?

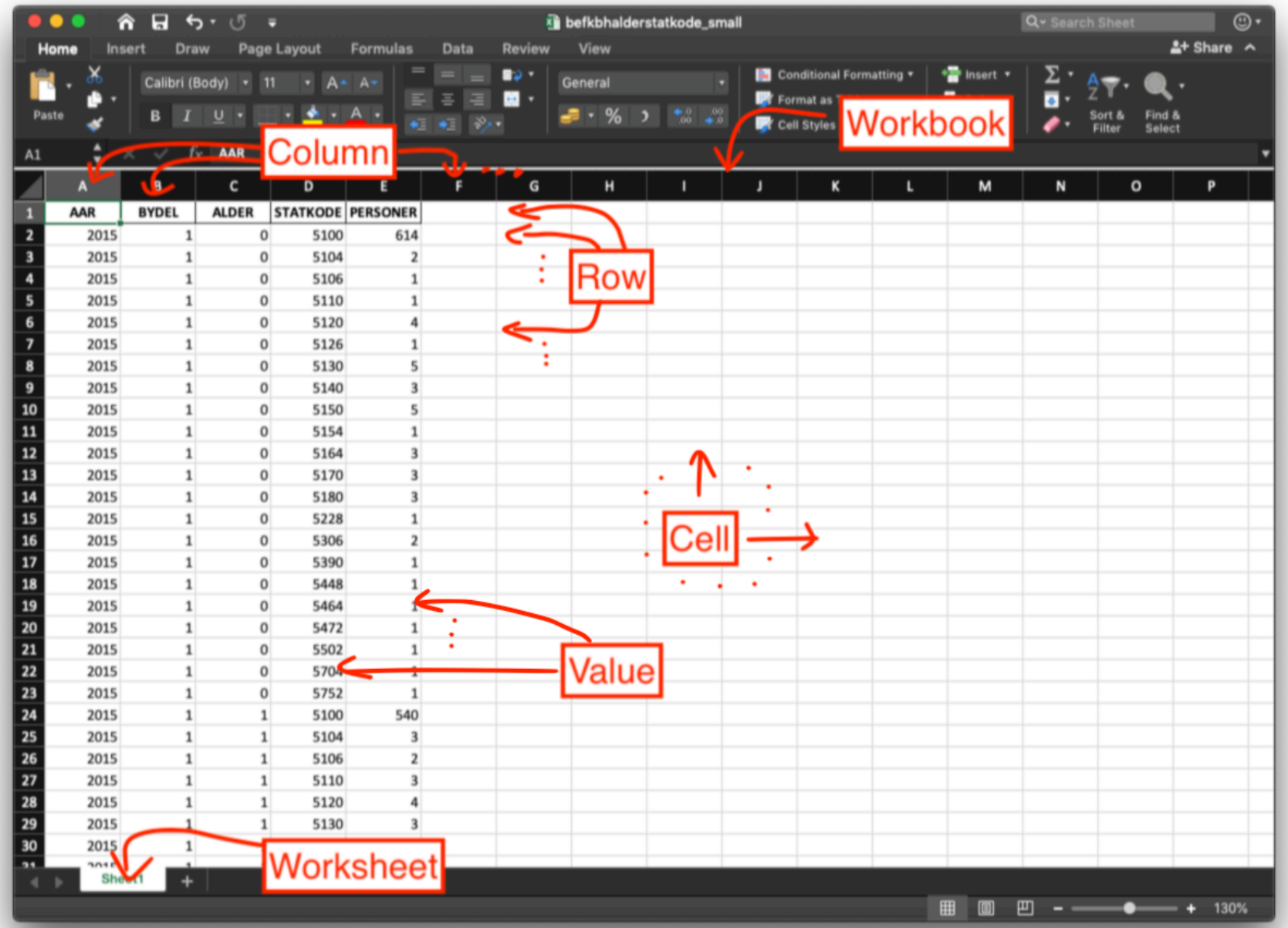
Describe what you see.



	A	B	C	D	E
1	AAR	BYDEL	ALDER	STATKODE	PERSONER
2	2015	1	0	5100	614
3	2015	1	0	5104	2
4	2015	1	0	5106	1
5	2015	1	0	5110	1
6	2015	1	0	5120	4
7	2015	1	0	5126	1
8	2015	1	0	5130	5
9	2015	1	0	5140	3
10	2015	1	0	5150	5
11	2015	1	0	5154	1
12	2015	1	0	5164	3
13	2015	1	0	5170	3
14	2015	1	0	5180	3
15	2015	1	0	5228	1
16	2015	1	0	5306	2
17	2015	1	0	5390	1
18	2015	1	0	5448	1
19	2015	1	0	5464	1
20	2015	1	0	5472	1
21	2015	1	0	5502	1
22	2015	1	0	5704	1
23	2015	1	0	5752	1
24	2015	1	1	5100	540
25	2015	1	1	5104	3
26	2015	1	1	5106	2
27	2015	1	1	5110	3
28	2015	1	1	5120	4
29	2015	1	1	5130	3
30	2015	1	1	5140	1

Object-oriented analysis

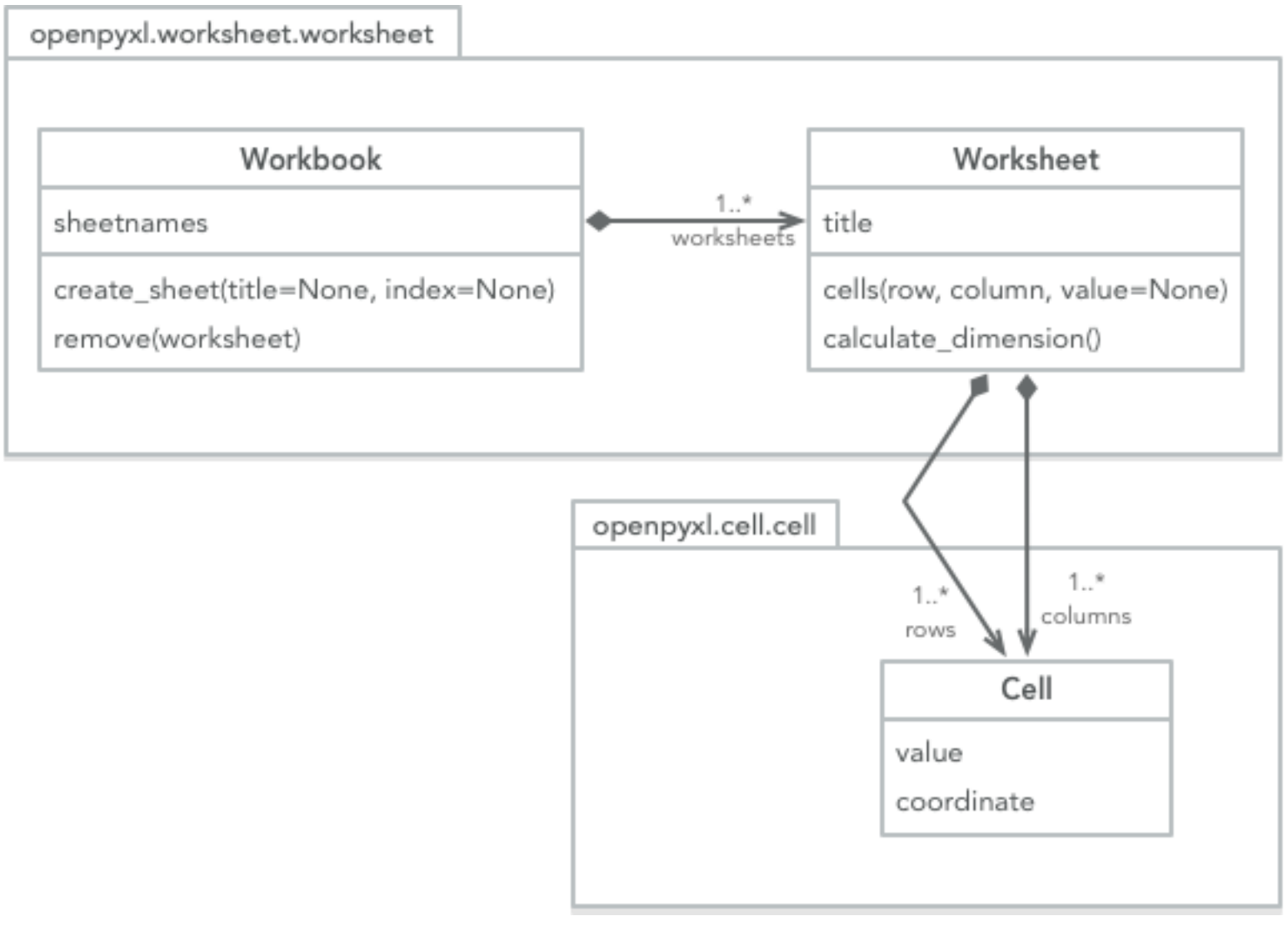
One of the first activities when writing object-oriented code is to identify what we are talking about (entities/classes), how these classes are related, and which behavior (methods) is supported by each of those.



	A	B	C	D	E
1	AAR	BYDEL	ALDER	STATKODE	PERSONER
2	2015	1	0	5100	614
3	2015	1	0	5104	2
4	2015	1	0	5106	1
5	2015	1	0	5110	1
6	2015	1	0	5120	4
7	2015	1	0	5126	1
8	2015	1	0	5130	5
9	2015	1	0	5140	3
10	2015	1	0	5150	5
11	2015	1	0	5154	1
12	2015	1	0	5164	3
13	2015	1	0	5170	3
14	2015	1	0	5180	3
15	2015	1	0	5228	1
16	2015	1	0	5306	2
17	2015	1	0	5390	1
18	2015	1	0	5448	1
19	2015	1	0	5464	1
20	2015	1	0	5472	1
21	2015	1	0	5502	1
22	2015	1	0	5704	1
23	2015	1	0	5752	1
24	2015	1	1	5100	540
25	2015	1	1	5104	3
26	2015	1	1	5106	2
27	2015	1	1	5110	3
28	2015	1	1	5120	4
29	2015	1	1	5130	3
30	2015	1			

Object-oriented design

If we were to draw an image in a formal language to represent the knowledge of the above annotations, it would likely look similar to the following:



This is an UML diagram. We are not going into detail about those but you will very likely meet them in your professional lives.

Object-oriented design in Python Code

An incomplete implementation of the previous UML diagram in Python code could look like this.

```
class Workbook:
    def __init__(self):
        self.sheetnames = []  # a list of strings
        self.worksheets = []  # a list of Worksheet objects

    def create_sheet(self, title=None, index=None):
        # ...

    def remove(self, workbook):
        # ...

class Worksheet:
    def __init__(self):
        self.title = ''  #  string for sheet name
        self.columns = ()  # tuple of cells
        self.rows = ()  # tuple of cells

    def cells(self, row, column, value=None):
        # ...

    def calculate_dimension(self):
        # ...

class Cell:
    def __init__(self):
        self.value = ''  #  string for sheet name
        self.coordinate = ''  # string with own coordinate
```

The Application Programming Interface (API)

The collection of accessible (callable) methods and properties forms the application programming interface (API).

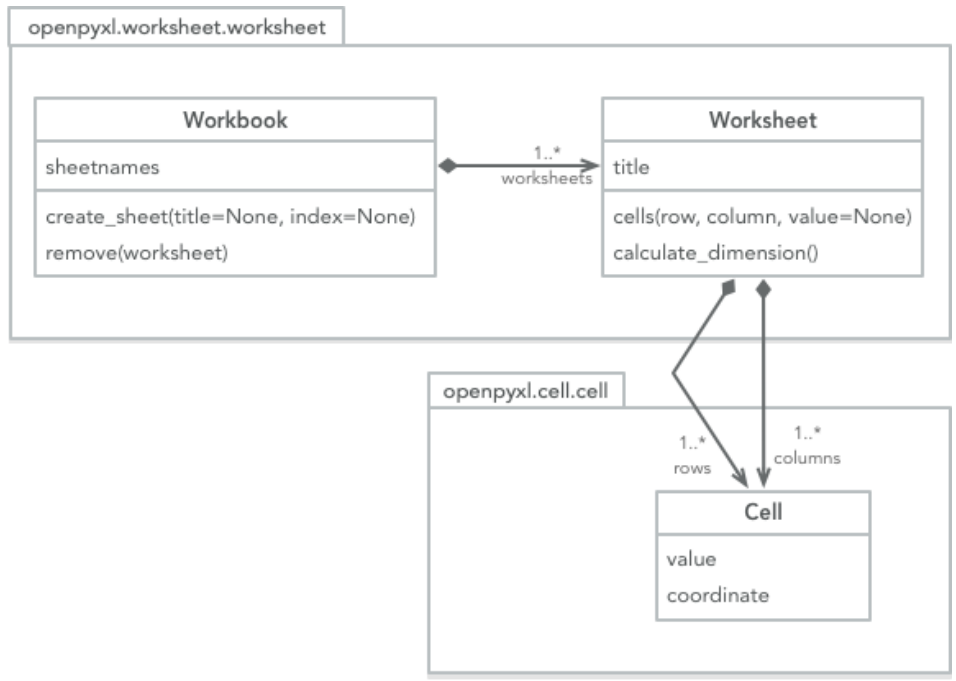
```
class Workbook:
    def __init__(self):
        self.sheetnames = []  # a list of strings
        self.worksheets = []  # a list of Worksheet objects

    def create_sheet(self, title=None, index=None):
        # ...

    def remove(self, workbook):
        # ...
class Worksheet:
    def __init__(self):
        self.title = ''  #  string for sheet name
        self.columns = ()  # tuple of cells
        self.rows = ()  # tuple of cells

    def cells(self, row, column, value=None):
        # ...

    def calculate_dimension(self):
        # ...
class Cell:
    def __init__(self):
        self.value = ''  #  string for sheet name
        self.coordinate = ''  # string with own coordinate
```



In your later careers you will meet colleagues, customers, contractors, etc. that talk about APIs. In this lecture, we will work with `openpyxl`, a Python library that allows you to read and write Microsoft Excel files.

Excel Documents

An Excel spreadsheet document is called a *workbook*. A single workbook is saved in a file with the `.xlsx` extension.

Each workbook can contain multiple sheets -also called *worksheets*. The sheet the user is currently viewing (or last viewed before closing Excel) is called the *active sheet*.

Each sheet has *columns* (addressed by letters starting at A) and *rows* (addressed by numbers starting at 1). A box at a particular column and row is called a cell. Each cell can contain a number or text value. The grid of cells with data makes up a sheet.

The Copenhagen population dataset

We will use a spreadsheet named `befkbhalderstatkode_small.xlsx` stored in the folder of this session.

The dataset contains statistics about Copenhagen's population corresponding in 2015.

Opening Excel Documents with `openpyxl`

- We will work with the document through a library called `openpyxl`. A 'library' is a collection of modules that can make certain things easy for us.
- You can start working with `openpyxl` by importing the `openpyxl` module like so:

```
import openpyxl
```

Once you have imported the openpyxl module, you will be able to use the openpyxl `.load_workbook()` function. You can get a list of all the sheet names in the workbook by calling the `get_sheet_names()` method.

In [17]:

```
import openpyxl


filename = 'befkbhalderstatkode_small.xlsx'
wb = openpyxl.load_workbook(filename)
print(wb)
```

```
<openpyxl.workbook.workbook.Workbook object at 0x1113b4978>
```

In [18]:

```
print(wb.sheetnames)
```

```
['Sheet1', 'Sheet2']
```

The `openpyxl.load_workbook()` function takes in the filename and returns a value of the workbook data type.

- This Workbook object represents the Excel file, a bit like how a File object represents an opened text file.
- Remember that `befkbhalderstatkode_small.xlsx` needs to be in the current working directory in order for you to work with it.

How to learn about the API of a new library like `openpyxl`?

- New libraries have tons of new features, including
  - Variables (like `string.ascii_letters`)
  - Functions (like `os.path.isfile`)
  - Classes (like `openpyxl.workbook.workbook.Workbook`)
- How do you know what variables, functions and classes are in there?!

- You go to the documentation:
  - https://lmddgtfy.net/?q=openpyxl (https://lmddgtfy.net/?q=openpyxl)

Getting Sheets from the Workbook

- Each sheet is represented by a Worksheet object, which you can obtain by passing the sheet name string to the `get_sheet_by_name()` workbook method.
- You can call the `get_active_sheet()` method of a Workbook object to get the workbook's active sheet. The active sheet is the sheet that's on top when the workbook is opened in Excel. Once you have the Worksheet object, you can get its name from the title attribute.

In [ ]:

```
import openpyxl


filename = 'befkbhalderstatkode_small.xlsx'
wb = openpyxl.load_workbook(filename)

sheet = wb['Sheet1']
print(sheet.title)
print(wb.active)
```

Getting Cells from the Sheets

Once you have a Worksheet object, you can access a Cell object by its name.

The Cell object has a value attribute that contains the value stored in that cell. Cell objects also have *row*, *column*, and *coordinate attributes* that provide location information for the cell.

In [19]:

```
cell = sheet['B1']
print(cell)
```

```
<Cell 'Sheet1'.B1>
```

We can get a 'cell' by accessing the exact cell 'B1' in the spreadsheet.

Notice we are not yet printing the contents, just verifying that this is indeed a cell.

In [20]:

```
print((cell.row, cell.column, cell.value))
```

```
(1, 2, 'BYDEL')
```

The row attribute gives us the integer `1`, the column attribute gives us `B`, and the coordinate attribute gives us 'Sepal width'.

In [21]:

```
1 print(sheet.cell(row=1, column=2))
```

```
<Cell 'Sheet1'.B1>
```

This is another way of doing the same. We access row 1 and column 2 ('B').

In [ ]:

```
1 print(sheet.cell(row=1, column=2) == cell)
```

In [ ]:

```
1 print(sheet.cell(row=1, column=2).value)
2 print(sheet['B1'].value)
3 print(cell.value)
```

Here, accessing the value attribute of our Cell object for cell `B1` gives us the string 'BYDEL', i.e., neighbourhood.

OpenPyXL will automatically interpret the values in cells and return them as as values of the correct type rather than strings.

Exercise!

1. Open the Befolknings data

   ```
   openpyxl.load_workbook(filename)
   ```

2. Store the sheet object in a variable
   - Remember you can get a sheet by doing:

     ```
     workbook["Sheet1"]
     ```

   - What is type of the variable?
3. Using the `[]` notation of the sheet, get the cell `C3`
4. Can you use the range-notation `:` from lists to get the cells from `A1` to `C3`?

Getting Rows and Columns from the Sheets

You can slice Worksheet objects to get all the Cell objects in a row, column, or rectangular area of the spreadsheet. Then you can loop over all the cells in the slice.

In [23]:

```
sheet['A1':'D4'][3][0]
```

Out[23]:

```
<Cell 'Sheet1'.A4>
```

```
sheet['A1':'D4']
```

Here, we specify that we want the Cell objects in the rectangular area from A1 to D4.

This tuple contains four tuples: one for each row, from the top of the desired area to the bottom.

Printing cells within rows

To print the values of each cell in the area, we use two `for` loops. The outer for loop goes over each row in the slice. Then, for each row, the nested for loop goes through each cell in that row.

In [24]:

```
for row_of_cell_objects in sheet['A1':'D4']:
    for cell_obj in row_of_cell_objects:
        print(cell_obj.coordinate, cell_obj.value)
    print('---------')
```

```
A1 AAR
B1 BYDEL
C1 ALDER
D1 STATKODE
---------
A2 2015
B2 1
C2 0
D2 5100
---------
A3 2015
B3 1
C3 0
D3 5104
---------
A4 2015
B4 1
C4 0
D4 5106
---------
```

Getting an entire column via its position in a list

In [ ]:

```
for cellObj in list(sheet.columns)[2]:
    print(cellObj.value)
```

Converting Between Column Letters and Numbers

To convert from letters to numbers, call the `openpyxl.utils.column_index_from _string()` function. To convert from numbers to letters, call the `openpyxl.utils.get_column_letter()` function.

In [ ]:

```
columns = ['A', 'B', 'C', 'D', 'E']

header = []
for column_name in columns:
    idx = column_name + '1'
    header.append(sheet[idx].value)

print(header)
```

In [ ]:

```
import openpyxl.utils as exutil

columns = ['A', 'B', 'C', 'D', 'E']

col_values = []
for column_name in columns:
    col_idx = exutil.column_index_from_string(column_name) - 1
    column = []
    for cell in list(sheet.columns)[col_idx][1:]:
        column.append(cell.value)

    col_values.append(column)

print(col_values)
```

Exercise!!!

- Write a program `report_all_nationalities.py`, which prints from which countries citicens lived in Copenhagen in 2015.
  - Let your program read data from the Excel file `befkbhalderstatkode_small.xlsx`.
  - *Hint:* Remember that you can read an Excel sheet as in the following:

```
filename = 'befkbhalderstatkode_small.xlsx'
wb = openpyxl.load_workbook(filename)
sheet = wb.get_sheet_by_name("Sheet1")
wb.get_active_sheet()
```

  - *Hint:* Use a `for` loop iterating over each row in the spreadsheet.
  - *Hint:* the value of the element in the fourth cell of the row contains the country code.
  - *Hint:* Make use of the module `stats_code.py`, which contains a variable with a dictionary `COUNTRY_CODES`. That is, after importing the module, you can lookup the description of a country code via something like the following, which should print `'Belgien'`:

```
belgien_code = stat_codes.COUNTRY_CODES['5126']
print(belgien_code)
```

- Write a small CLI program that counts how many people from a certain nation lived in Copenhagen in 2015.
  - Call your program `count_nationalities.py`
  - Let it read data from the Excel file `befkbhalderstatkode_small.xlsx`
  - Let it consume an argument from the command-line, which says which nationality you want to count. For example, running `count_nationalities.py` with the CLI argument `5126` -the country code for Belgien- should print `273`:

```
$ python count_nationalities.py 5126
273
```

  - *Hint:* Use a `for` loop iterating over each row in the spreadsheet.
  - *Hint:* When the value of the element in first cell of the row is `2015` and the value of the fourth cell of the row is the given country code then you want to use the value of the last cell of the row for counting.

In case you are interested, the Economist published an article about Python recently, saying amongst others that some companies that use Excel extensively now started training their staff in programming Python.

"Python has brought computer programming to a vast new audience"
https://www.economist.com/science-and-technology/2018/07/21/python-has-brought-computer-programming-to-a-vast-new-audience (https://www.economist.com/science-and-technology/2018/07/21/python-has-brought-computer-programming-to-a-vast-new-audience)