Good practices 1: Modularization and Code Style

Miguel Ángel Latre

latre@unizar.es

Advanced Information Systems Laboratory
Universidad Zaragoza

TWEED Project, SC1

Justification

- If you know how to write a 100 LOC (lines of code) program, you might think that writing a 100 KLOC program consists of doing the same thing a thousand times in a row.
  - This is not the case.
- Knowing how to modularize the code is basic to be able to deal with large, well-organized, understandable and maintainable programs.

Objectives

- Facilitate code maintenance as the software grows in size
- Ways to achieve it
  - Appropriate modularization
    - Procedures and functions
    - Classes and objects
    - Modules and packages
  - Coding style
    - Documentation
    - Uniformity
    - Static analysis tools
    - Refactoring

Outline

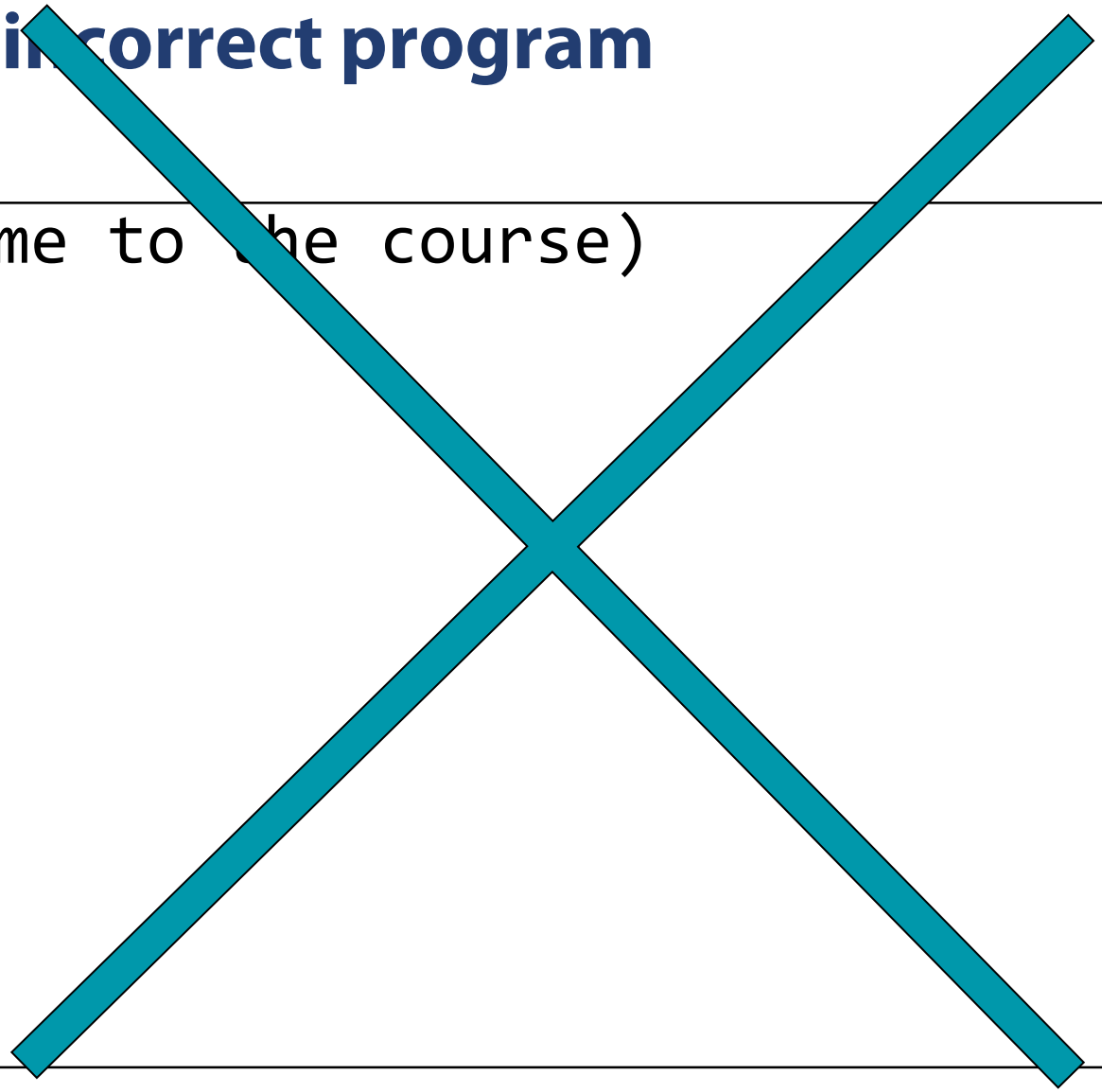
- Properties of a program
- Modularization
- Code Style

Properties of a program

- **Essential**
  - Correction
- **Desirable**
  - Readability
  - Generality
  - Reusability
  - Simplicity
  - Efficiency
  - Machine and language independence
  - Robustness
  - …

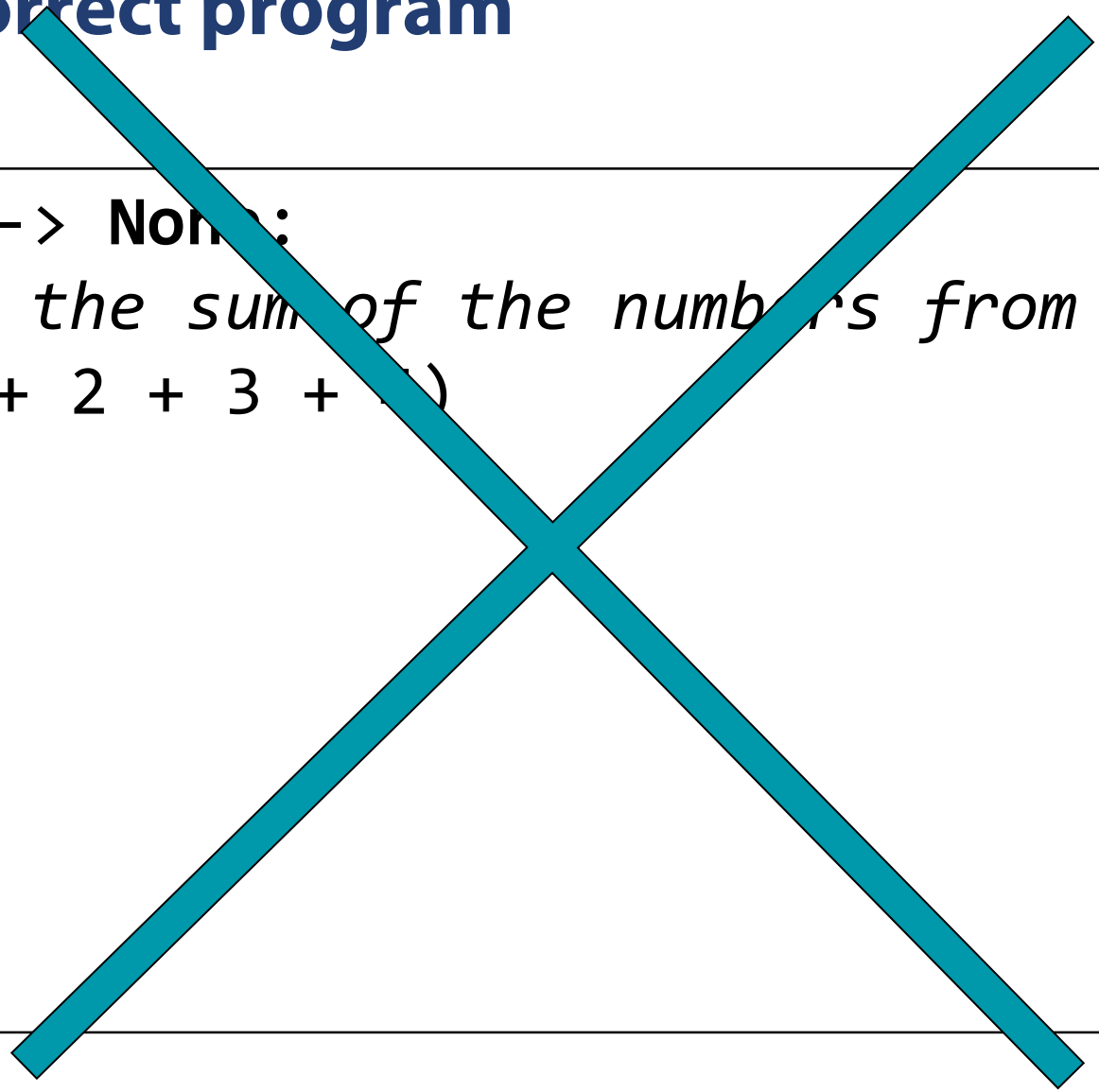
Syntactically incorrect program

```
print("Welcome to the course)
```



Formally incorrect program

```
def write() -> None:
    """Write the sum of the numbers from 1 to 5."""
    print(1 + 2 + 3 + 4 + 5)
```



Properties of an algorithm

- **Essential**
  - Correction
- **Desirable**
  - **Readability**
  - Generality
  - Reusability
  - Simplicity
  - Efficiency
  - Machine and language independence
  - Robustness

Unreadable program in Python

```
def a(b):
    if len(b.replace('/', '-').replace('.', '-').split('-')) != 3: raise ValueError('Error #1
    d = int(b.replace('/', '-').replace('.', '-').split('-')[2])
    if d <= 1582: raise ValueError('Error #2
    e = int(b.replace('/', '-').replace('.',      .split('-')[1])
    if e < 1 or e > 12: raise ValueError('Error
    f = 31
    if e == 2:
        if d % 400 == 0: f = 29
        elif d % 100 == 0: f = 28
        elif d % 4 == 0: f = 29
        else: f = 28
    elif e in (4, 6, 9, 11):
        f = 30
    g = int(b.replace('/', '-').replace('.', '-').split('-')[0])
    if g < 1 or g > f:
        raise ValueError('Error #4')
    g += 1
    if g > f:
        g = 1
        e += 1
        if e > 12:
            e = 1
            d += 1
    return '-'.join([str(g), str(e), str(d)])
```

Properties of an algorithm

- **Essential**
  - Correction
  - **Readability**
- **Desirable**
  - Generality
  - Reusability
  - Simplicity
  - Efficiency
  - Machine and language independence
  - Robustness

Properties of an algorithm

- **Essential**
  - Correction
  - Readability
- **Desirable**
  - **Generality**
  - **Reusability**
  - Simplicity
  - Efficiency
  - Machine and language independence
  - Robustness

Generality

```
def sum_1_to_100() -> int:
    """Return the sum of integers between 1 and 100.
    """

    result = 0
    for i in range(1, 100 + 1):
        result += i
    return result
```

Generality

```
def sum_interval(start: int, end: int) -> int:
    """Return the sum of integers between start and
    end.
    """

    result = 0
    for i in range(start, end + 1):
        result += i
    return result
```

Properties of an algorithm

- **Essential**
  - Correction
  - Readability
- **Desirable**
  - Generality
  - Reusability
  - **Simplicity**
  - Efficiency
  - Machine and language independence
  - Robustness

Simplicity

```
def sum_interval(start: int, end: int) -> int:
    """Return the sum of integers between start and
    end.
    """

    return sum(range(start, end + 1))
```

Properties of an algorithm

- **Essential**
  - Correction
  - Readability
- **Desirable**
  - Generality
  - Reusability
  - Simplicity
  - **Efficiency**
  - Machine and language independence
  - Robustness

Efficiency

```
def sum_interval(start: int, end: int) -> int:
    """Returns the sum of integers between start and
    end.
    """

    return (start + end) * (end - start + 1) // 2
```

Properties of an algorithm

- **Essential**
  - Correction
  - Readability
- **Desirable**
  - Generality
  - Reusability
  - Simplicity
  - Efficiency
  - Machine and language **independence**
  - Robustness

Properties of an algorithm

- **Essential**
  - Correction
  - Readability
- **Desirable**
  - Generality
  - Reusability
  - Simplicity
  - Efficiency
  - Machine and language independence
  - **Robustness**

Robustness

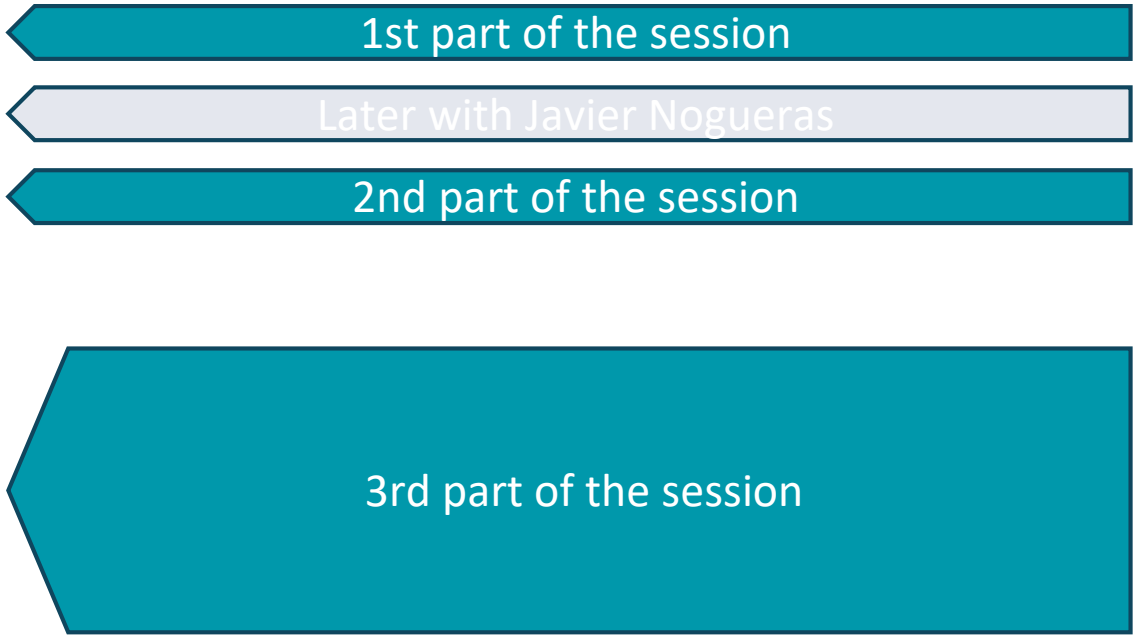
```
Type an integer: an integer
Traceback (most recent call last):
  File "e:\ex_04_robust.py", line 4, in <module>.
    n = int(input('Type an integer: '))
ValueError: invalid literal for int() with base 10:
'an integer'.
```

Properties of an algorithm

- **Essential**
  - Correction
  - Readability
- **Desirable**
  - General
  - Reusability
  - Simplicity
  - Efficiency
  - Machine and language independence
  - Robustness

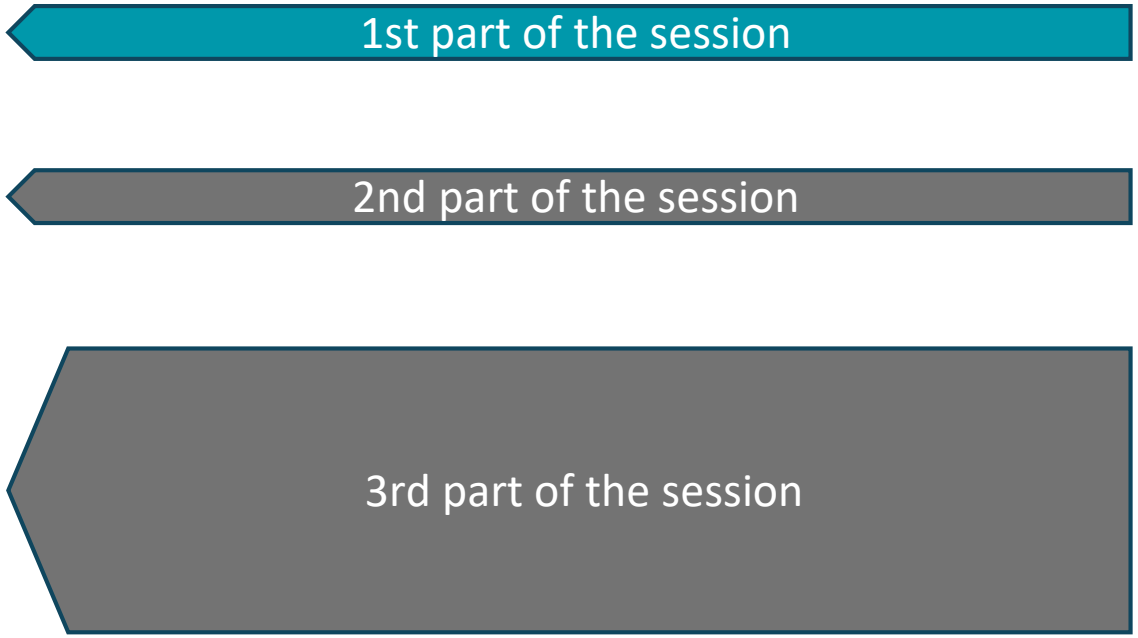
Ways to facilitate maintenance

- Modularization
  - Procedures and functions
  - Classes and objects
  - Modules and packages
- Code style
  - Documentation
  - Uniformity in coding
  - Static analysis tools
  - Refactoring



Ways to facilitate maintenance

- Modularization
  - Procedures and functions
  - Classes and objects
  - Modules and packages
- Code style
  - Documentation
  - Uniformity in coding
  - Static analysis tools
  - Refactoring



Modularization

- Elements
  - **Procedure**: abstraction of an instruction
  - **Function**: abstraction of an expression
  - **Class**: data abstraction + behaviour
  - **Packages** and **libraries**: logical and physical organization of code
- Aspects to take into account
  - Cohesion
  - Coupling
- Objectives
  - Readability, maintainability and reusability

Non-modular example

$$\binom{n}{m}$$

```
# I want the binomial coefficient C(n, m)
num = 1
for i in range(1, n + 1):
    num = num * i
den1 = 1
for i in range(1, m + 1):
    den1 = den1 * i
den2 = 1
for i in range(1, n - m + 1):
    den2 = den2 * i
comb = num // (den1 * den2)
```

Modular example

$$\binom{n}{m}$$

```
# I want the binomial coefficient C(n, m)
def factorial(n: int) -> int:
    """If n ≥ 0, return n!"""
    res = 1
    for i in range(1, n + 1):
        res = res * i
    return res


def binomial(n: int, m: int) -> int:
    """If n ≥ m ≥ 0, return C(n m)"""
    return factorial(n) // (factorial(m) * factorial(n - m))
```

$$\binom{n}{m} = \frac{n!}{m!\,(n-m)!}$$

Modular and more efficient example

... but not so simple and not so readable

$$\binom{n}{m}$$

```
# I want the binomial coefficient C(n, m)
def binomial(n: int, m: int) -> int:
    """If n ≥ m ≥ 0, return C(n m)"""
    res = 1
    for i in range (1, m + 1):
        res = res * (n + 1 - i) // i
    return res
```

$$\binom{n}{m} = \prod_{i=1}^{m} \frac{n+1-i}{i}$$

Structuring your code

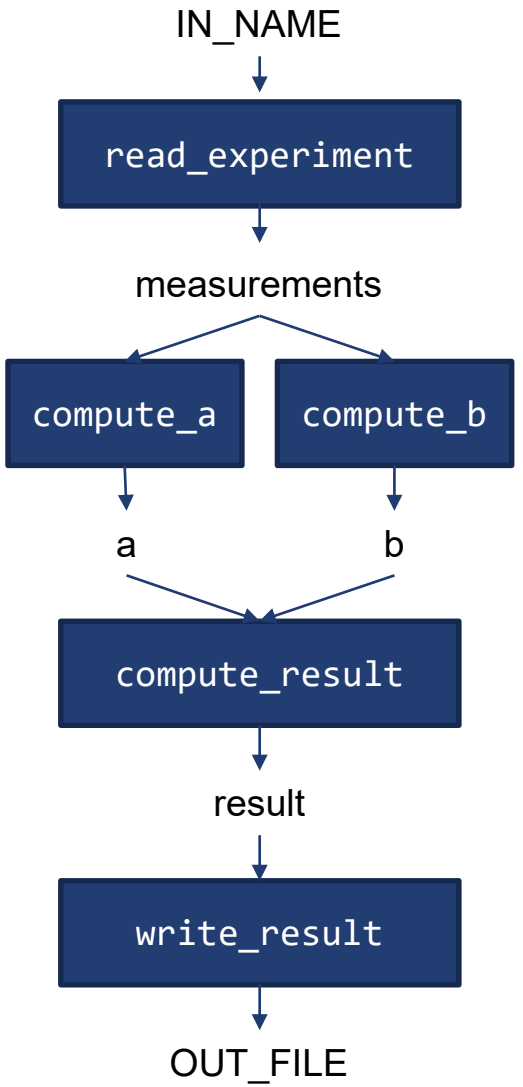
- Separation of concerns
- Levels of abstraction
- Single responsibility principle

Separation of concerns

- **Separation of Concerns** (SoC) is a fundamental design principle in software engineering that involves **dividing a program into distinct sections, each addressing a separate concern**. A concern is a specific aspect of functionality, behaviour, or responsibility within the system.
- A piece of code should be split into multiple parts, each of them dealing with one aspect of the workflow. This makes the code more readable, and also more reusable, since the different parts could be mixed if their functionality were clearly separated.

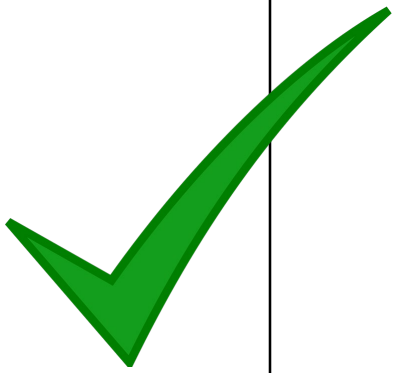
Separation of concerns

```
def main():
    measurements = read_experiment(IN_NAME)

    a = compute_a(measurements)
    b = compute_b(measurements)
    result = compute_result(a, b)

    write_result(result, OUT_FILE)
```



Modularity and data flow

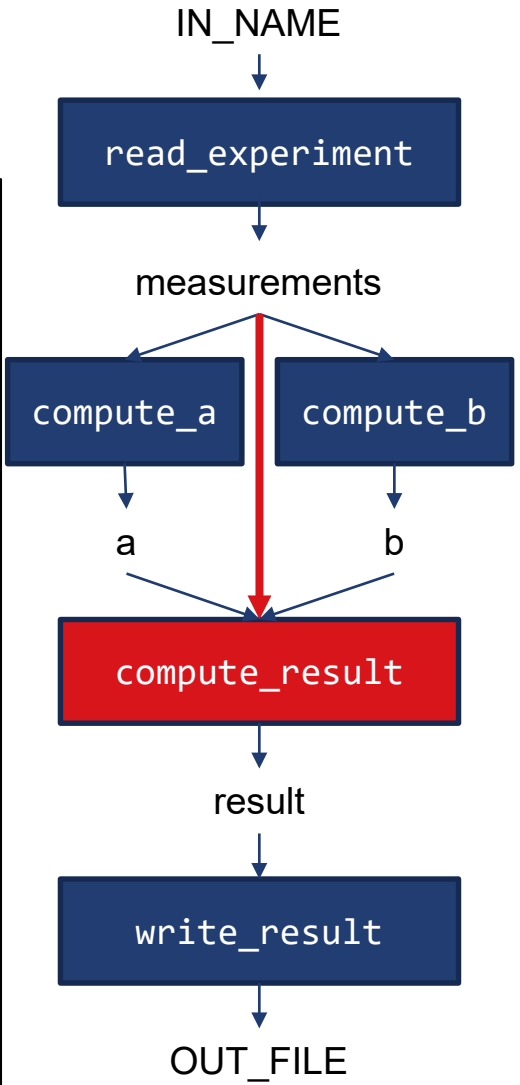
```
def read_experiment(input_file):
    ...
    return measurements

def compute_a(measurements):
    ...
    return a

def compute_b(measurements):
    ...
    return b

def compute_result(a, b):
    ...
    return result

def write_result(result, output_file):
    ...
```



Modularity and data flow

Bad practice: increasing coupling

```
def compute_result(a, b, measurements):
    ...
    return result

def main():
    measurements = read_experiment(FILE_NAME)

    a = compute_a(measurements)
    b = compute_b(measurements)
    result = compute_result(a, b, measurements)

    write_result(result, OUT_FILE)
```

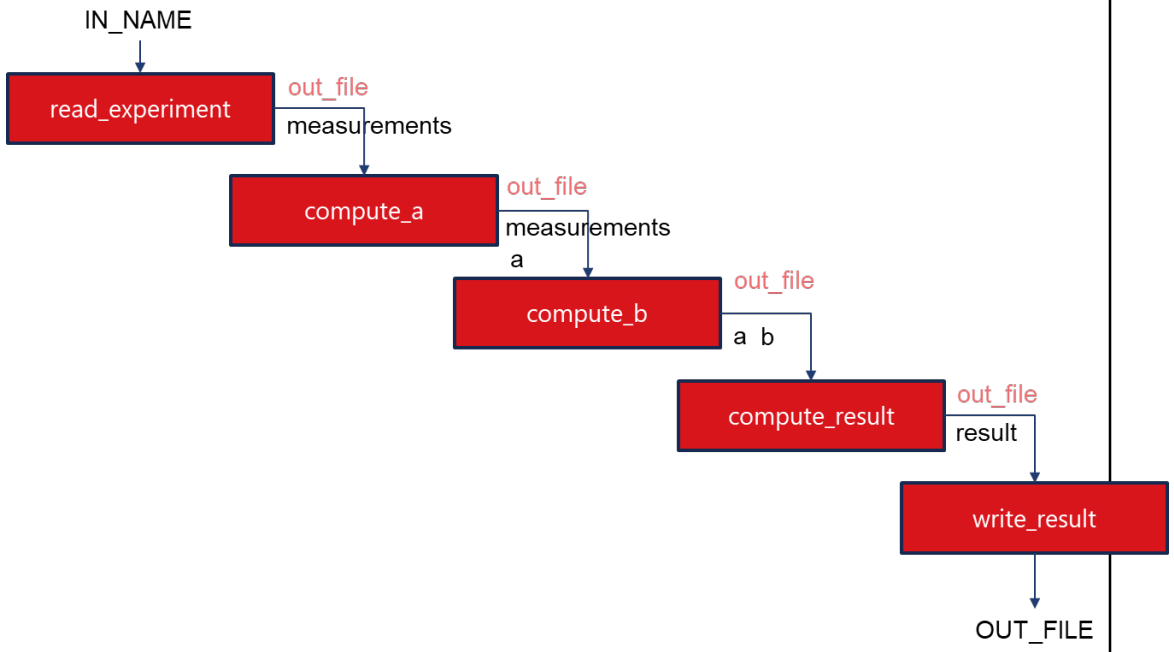


Source: Imperial College Research Computing Service. *Essential Software Engineering for Researchers.* Imperial College London. 2024. https://imperialcollegelondon.github.io/grad_school_software_engineering_course/

Modularity and data flow

Bad practice: nesting (hiding) invocations

```
def read_experiment(input_file, output_file):
    measurements = ...
    compute_a(measurements, output_file)

def compute_a(measurements, output_file):
    a = ...
    compute_b(measurements, a, output_file)

def compute_b(measurements, a, output_file):
    b = ...
    compute_result(a, b, output_file)

def compute_result(a, b, output_file):
    result = ...
    write_result(result, output_file)

def write_result(output_file):
    ...

def main():
    read_experiment(IN_FILE, OUT_FILE)
```



Levels of abstraction

- **Abstraction** is a concept in computer science and software engineering that involves **generalizing concrete details to focus on more important, generic aspects**. It helps in managing complexity by hiding implementation details and exposing only the necessary parts.

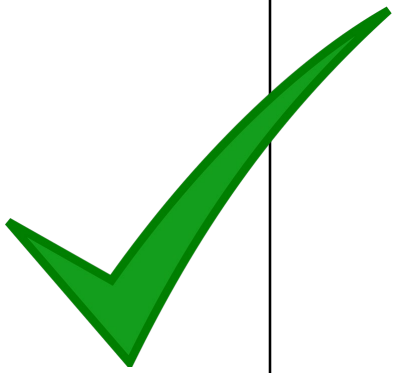
Levels of abstraction

Bad practice: exposing details

```
def analyse(input_path):
    data = load_data(input_path)

    # special fix for bad data point
    data[0].x["january"] = 0

    results = process_data(data)
    return results
```

Levels of abstraction

```
def analyse(input_path):
    data = load_data(input_path)

    cleaned_data = clean_data(data)

    results = process_data(cleaned_data)
    return results
```



Single responsibility principle

- The **Single Responsibility Principle** (SRP) states that **a class or module should have only one reason to change**, meaning it should have only one responsibility or purpose. This principle helps in creating more maintainable, reusable, and flexible software designs.
- This goes into a deeper level than the abstractions described before, and it is probably the one harder to do right, besides its apparent simplicity. It literally means that each function or method should do one thing only. This typically means also that is short, easier to read, to debug and to re-use.

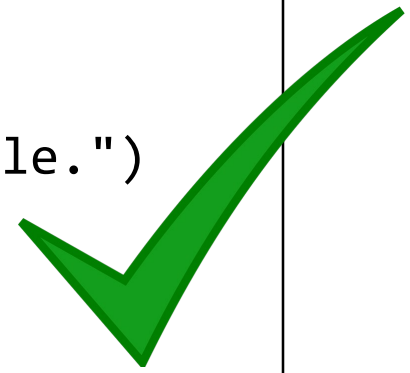
Single responsibility principle

Bad practice: doing too many things

```
def load_data(filename):
    if filename.suffix == ".xlsx":
        data = pd.read_excel(filename, usecols="A:B")
    elif filename.suffix == ".csv":
        data = pd.read_csv(filename)
        data["datetime"] = data["date"] + " " + data["time"]
        data = data.drop(columns=["date", "time"])
    else:
        raise RuntimeError("The file must be Excel or CSV.")

    assert data.columns == ["datetime", "value"]
    return data
```

Single responsibility principle

```
def load_data(filename):
    """Select how to load the data based on the file extension."""
    if filename.suffix == ".xlsx":
        data = load_excel(filename)
    elif filename.suffix == ".csv":
        data = load_csv(filename)
    else:
        raise RuntimeError("The file must be an Excel or a CSV file.")
    return data


def load_excel(filename):
    """Loads an Excel file, picking only the first 2 columns."""
    data = pd.read_excel(filename, usecols="A:B")
    validate(data)
    return data
```

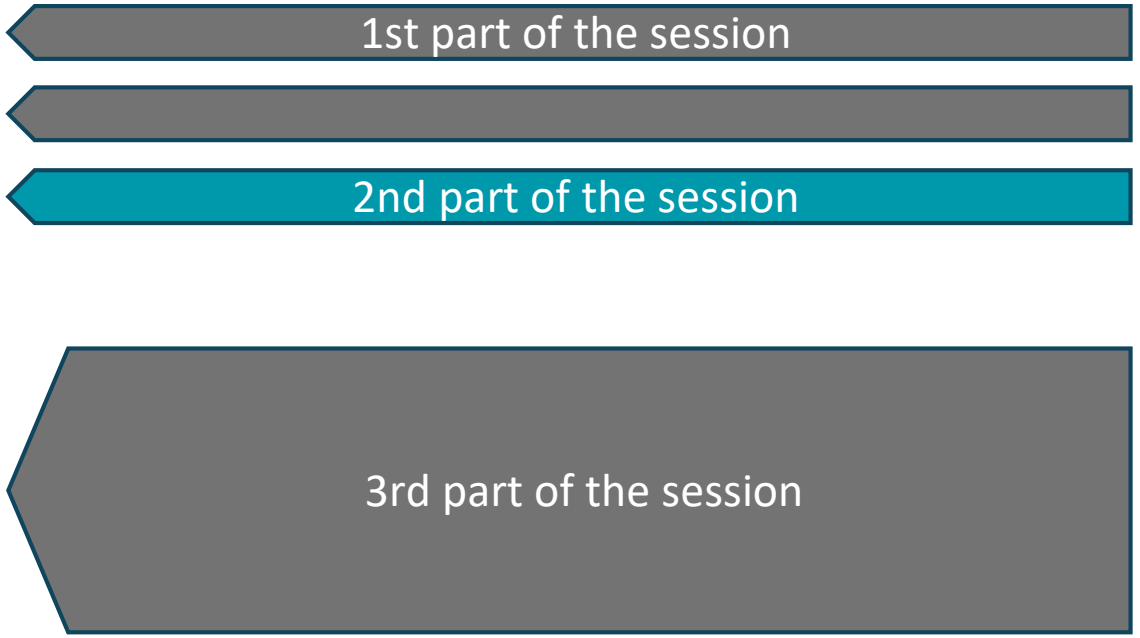


Single responsibility principle

```
def load_csv(filename):
    """Loads a CSV file, combining date and time."""
    data = pd.read_csv(filename)
    data["datetime"] = data["date"] + " " + data["time"]
    data = data.drop(columns=["date", "time"])
    validate(data)
    return data

def validate(data):
    """Checks that the data has the right structure."""
    assert data.columns == ["datetime", "value"]
```

Ways to facilitate maintenance

- Modularization
  - Procedures and functions
  - Classes and objects
  - Modules and packages
- Code style
  - Documentation
  - Uniformity in coding
  - Static analysis tools
  - Refactoring



Modules and packages

- Mechanism for grouping and organizing the code
- Module:
  - Group of related definitions and declarations,
    such as classes, constants, functions and variables.
  - In Python, it is a file with extension ".py".
    The module name is the name of the file (without the ".py" extension).
- Package:
  - Group of related modules.
  - Allows hierarchical structuring of the modules.
  - Physically, it is a directory of the file system.

Module example

prime.py

```
def is_prime(n: int) -> bool:
    if n < 2:
        return False
    prime = True
    i = 2
    while prime and i < n:
        if n % i == 0:
            prime = False
        i += 1
    return prime


...
```

main.py

```
import prime

print(prime.is_prime(1))
print(prime.is_prime(2))
print(prime.is_prime(3))
print(prime.is_prime(4))
print(prime.is_prime(5))
```

```
from prime import is_prime

print(is_prime(1))
print(is_prime(2))
print(is_prime(3))
print(is_prime(4))
print(is_prime(5))
```

main_alt.py

Criteria for grouping code into modules

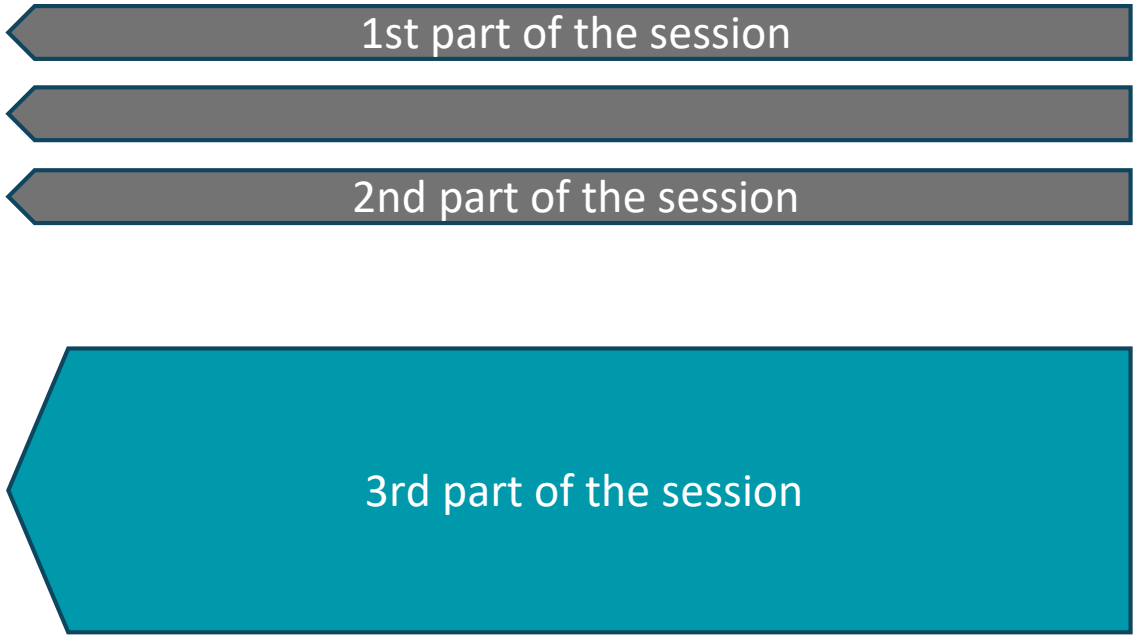
- By common functionality: functions that perform related tasks
  - User interaction
  - Data input from file and data output to file
  - Running models or simulations
  - Other sets of related functions
  - Data structures and related functions (in particular, classes)
- Coherence and Cohesion
  - Establish a well-defined responsibility for the module.
- Size and Readability
  - When modules becomes too large, split them into smaller modules.

Criteria for grouping code into packages

- Criteria common to those of the modules
- Also taking advantage of its possibilities of:
  - Hierarchical organization
  - Scalability
    - They allow the addition of new modules without disorganizing the existing structure.
  - Independent development and testing.

Ways to facilitate maintenance

- Modularization
  - Procedures and functions
  - Classes and objects
  - Modules and packages
- Code style
  - Documentation
  - Uniformity in coding
  - Static analysis tools
  - Refactoring



Documentation

- *Code is written once but read many times.*
  - It is not only programmed for the computer, but also for other people.
  - The code itself is the best way to document a program.
  - The code should be pleasant to read, another form of academic communication.
- Who do you document for?
  - For yourself: you are the person most likely to have to read your code and comments. Maybe in a week or maybe in six months.
  - For someone on your team or someone else who will probably have a similar level of experience and will be trying to do something similar.
  - Try to help the person reading the code understand what you did and why.

Documentation

- It is preferable to embed documentation in the code rather than write explicit comments:

```
def rep(n, m):
    # gives out candy to children
    # n is the number of candies
    # m is the number of children
    return n // m
```

```
def distribute(candies, children):
    return candies // children
```


Source: Construction and Design. *Research Software Engineering with Python*

Documentation

- Avoid superfluous comments:

```
# Increment the counter
counter = counter + 1

# Loop over elements
for element in array:
    # printing the element
    print(element)
```

```
counter = counter + 1

for element in array:
    print(element)
```

Documentation

- *Type hints*
  - Available since Python 3.5

```
def count_by_sender(file):
    # returns a dictionary of string keys and int values
    ...
```

```
def count_by_sender(file: TextIOWrapper) -> dict[str, int]:
    ...
```

Documentation

- Docstrings

```
def next_day(date: str) -> str:
    """Return a string representing one day after the argument date.

    Args:
        date: a string representing a date in the form 'day-month-year'.

    Returns:
        A string representing one day after the given date
    """
    return ...
```

- Example: https://google.github.io/styleguide/pyguide.html#38-comments-and-docstrings

Uniformity in coding

- Many ways to write the same code:

```
import species


def AddToReaction(aName, aReaction):
    lSpecies = species.Species(aName)
    aReaction.append(lSpecies)
```

```
from species import Species


def add_to_reaction(name, reaction):
    reaction.append(Species(name))
```


Source: Construction and Design. *Research Software Engineering with Python*

PEP 8 - Style Guide for Python Code

- Code Lay-out
  - Indentation: Use 4 spaces per indentation level.
  - Tabs or Spaces: Spaces
  - Maximum Line Length: Limit all lines to a maximum of 79 characters.
  - Should a Line Break Before or After a Binary Operator? Before
  - Blank Lines
    - Surround top-level function and class definitions with two blank lines.
    - Method definitions inside a class are surrounded by a single blank line.
  - Source File Encoding
    - UTF-8
    - All identifiers in the Python standard library MUST use ASCII-only identifiers, and SHOULD use English words wherever feasible.
  - Imports: on separate lines; at the top of the file.

Source: Guido van Rossum, Barry Warsaw, Alyssa Coghlan. "PEP 8 - Style Guide for Python Code." *Python Enhancement Proposals*. 2013. https://peps.python.org/pep-0008/

PEP 8 - Style Guide for Python Code

- String Quotes
  - Does not take sides
- Whitespace in Expressions and Statements
  - `a = 1`
- When to Use Trailing Commas
- Comments
- Naming Conventions
  - Variables, functions, methods, parameters: `snake_case`
  - Classes: `CamelCase`
  - Constants: `UPPER_CASE_WITH_UNDERSCORES`
- Programming Recommendations

Source: Guido van Rossum, Barry Warsaw, Alyssa Coghlan. "PEP 8 - Style Guide for Python Code." *Python Enhancement Proposals*. 2013. https://peps.python.org/pep-0008/

Code formatters

- They ensure that the code style in a project is homogeneous, even if several people contribute to it.
- Black
  - https://pypi.org/project/black/
  - https://black.readthedocs.io/en/stable/the_black_code_style/current_style.html
  - https://marketplace.visualstudio.com/items?itemName=ms-python.black-formatter

Linters

- Indicate deviations from the code style
- Avoid common mistakes:
  - Code that cannot be executed
  - Code with undefined behaviour
  - Variables that are never used
  - …
- Detect smells in the code
  (patterns that are bad programming practices).
- Encourage proper documentation of the code
- …

Source: Cottonbro studio. *Woman Using Lint Remover.* Pexels. *https://www.pexels.com/photo/woman-using-lint-remover-6865185/*

Linters

- Pyright
  - https://github.com/microsoft/pyright
  - Pylance: https://marketplace.visualstudio.com/items?itemName=ms-python.vscode-pylance
- Pylint
  - https://pylint.readthedocs.io/en/stable/
  - https://marketplace.visualstudio.com/items?itemName=ms-python.pylint
- Flake 8
  - Wrapper for PyFlakes, pycodestyle, and Ned Batchelder's McCabe script
  - https://flake8.pycqa.org/en/latest/
  - https://marketplace.visualstudio.com/items?itemName=ms-python.flake8

Linters
Boolean smells

```
def pos(a: Number) -> bool:
    if a > 0:
        return True
    else:
        return False

def print_conditionally(
    message: str,
    condition: bool,
) -> None:
    if condition == True:
        print(message)
```

```
def pos(a): Number) -> bool:
    return a > 0



def print_conditionally(
    message: str,
    condition: bool,
) -> None:
    if condition:
        print(message)
```

The Zen of Python

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Source: Tim Peters. "PEP 20 - The Zen of Python." *Python Enhancement Proposals*. 2004. https://peps.python.org/pep-0020/

Exercise 1

- https://github.com/miguel-latre/rse-sesion-02
- The code of the `next_date` function in the file `pb_01_next_day.py`, given a string representing a date, returns another string representing the date corresponding to the next day.
- We are asked for a similar function that, given a string representing a date, returns another string representing the date corresponding to the previous day.
- Before solving the problem, we will have to rewrite the code to increase its modularity, readability and reusability.

Exercise 1

- Code of `pb_01_next_day.py` is documented with many examples of use:

```
>>> next_date('1-7-2024')
'2-7-2024'
```

- You can run them with doctest from the command line:

```
python -m doctest pb_01_next_day.py -v
```

- Or by running directly the script (look at the code of the main() function):

```
python -u pb_01_next_day.py
```

- In Visual Studio Code, you may also install the Python DoctestBtn extension:
  - https://marketplace.visualstudio.com/items?itemName=NoahSyn10.pydoctestbtn

Exercise 2

- Python comes with "batteries included".
  - There are packages that solve the problem solved in exercise 1 better than us.
- Use the `datetime` module to solve the problem
  - But keep compatibility with the `next_day` function already implemented.

Exercise solutions

- Refactor of the next_date function:
  - https://iaaa.es/courses/2025-TWEED/mod-code-style-exercises/pb_01_next_day_ok.py
- Solution with the previous_day function:
  - https://iaaa.es/courses/2025-TWEED/mod-code-style-exercises/pb_01_next_prev_day_ok.py

To learn more...

- Imperial College Research Computing Service. *Essential Software Engineering for Researchers*. Imperial College London. 2024. https://imperialcollegelondon.github.io/grad_school_software_engineering_course/
  - A short course on software engineering for research, focusing on best practices and testing.
- Miscellaneous. *Research Software Engineering with Python*. The Alan Turing Institute. 2024. https://alan-turing-institute.github.io/rse-course/html/index.html
  - Course on software engineering for research, focusing on Python
- Guido van Rossum, Barry Warsaw, Alyssa Coghlan. "PEP 8 - Style Guide for Python Code." *Python Enhancement Proposals*. 2013. https://peps.python.org/pep-0008/