UNIVERSITÄT
BERN

10. Structuring a coding project for data science

Athina Tzovara

University of Bern

Athina.Tzovara@unibe.ch

Hallmarks of good scientific software

Lecture 9

version control

automated testing

self-reporting analyses

pipelining Today!

Lecture 8

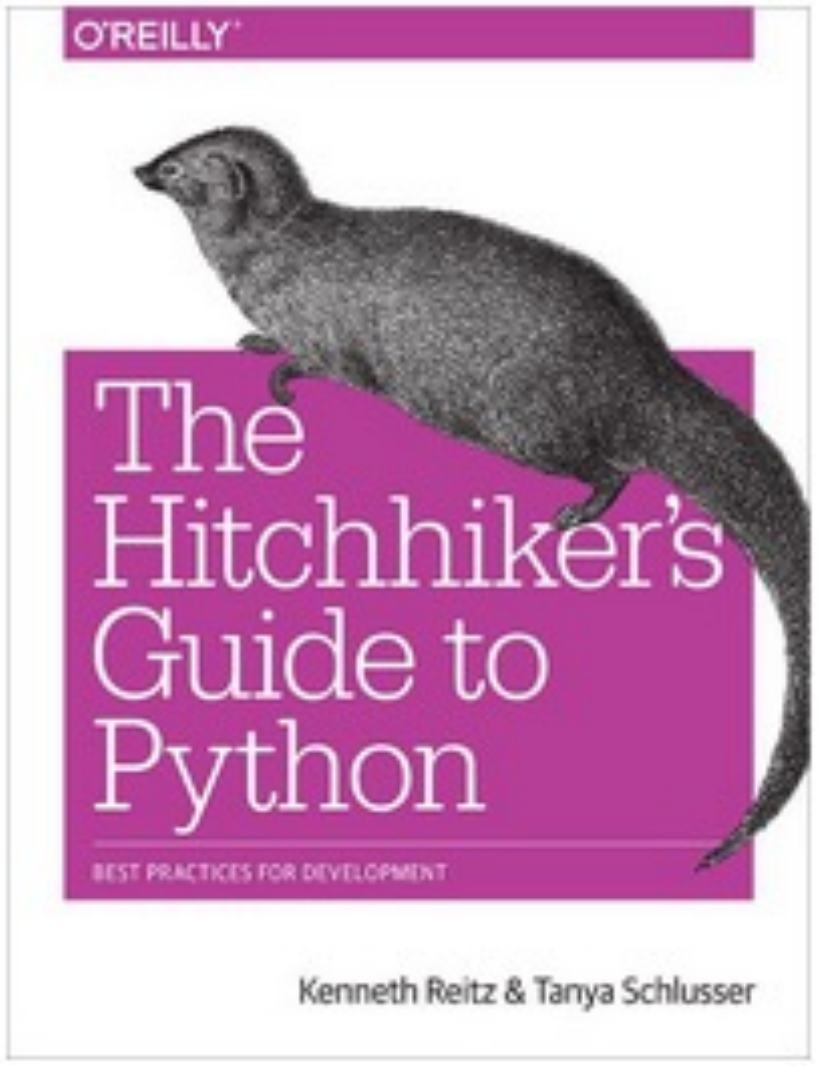
documentation

containerization

Source: https://grst.github.io/bioinformatics/2020/07/16/hallmarks-scientific-software.html

Main resource for today

https://docs.python-guide.org/



Outline for today

1. Coding principles for writing efficient code
2. Coding styles
3. Source Code Analysis

Structuring a project

Decisions that we make when working on a project:

- How does our project best meet its objective?

- How do we write clean and effective code?

Which functions go into which modules?

How do data flow through our project?

Which functions can be grouped together? Isolated?

Naming conventions?

Structure of Code in Python

Python naturally encourages structured code (e.g. Modules)

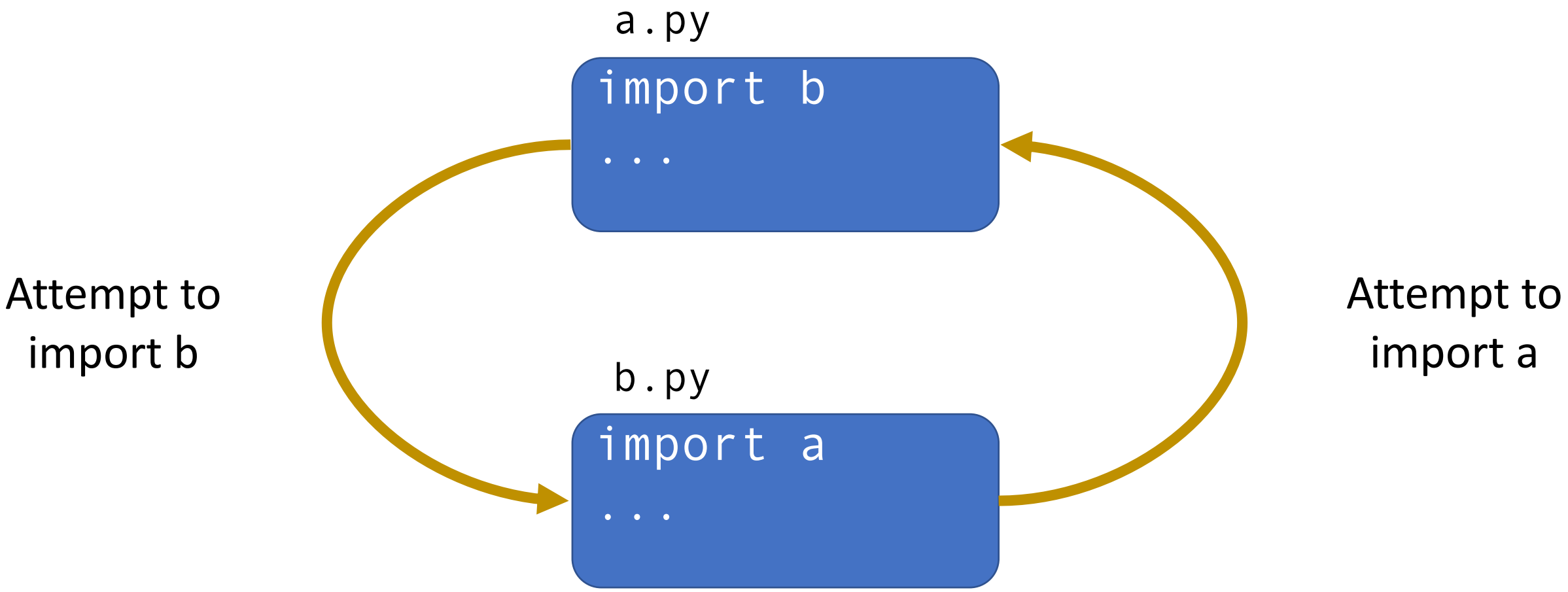
Not many constraints; importing modules is intuitive

Architectural questions: crafting different parts of a project and their interactions

What are signs of poor code?

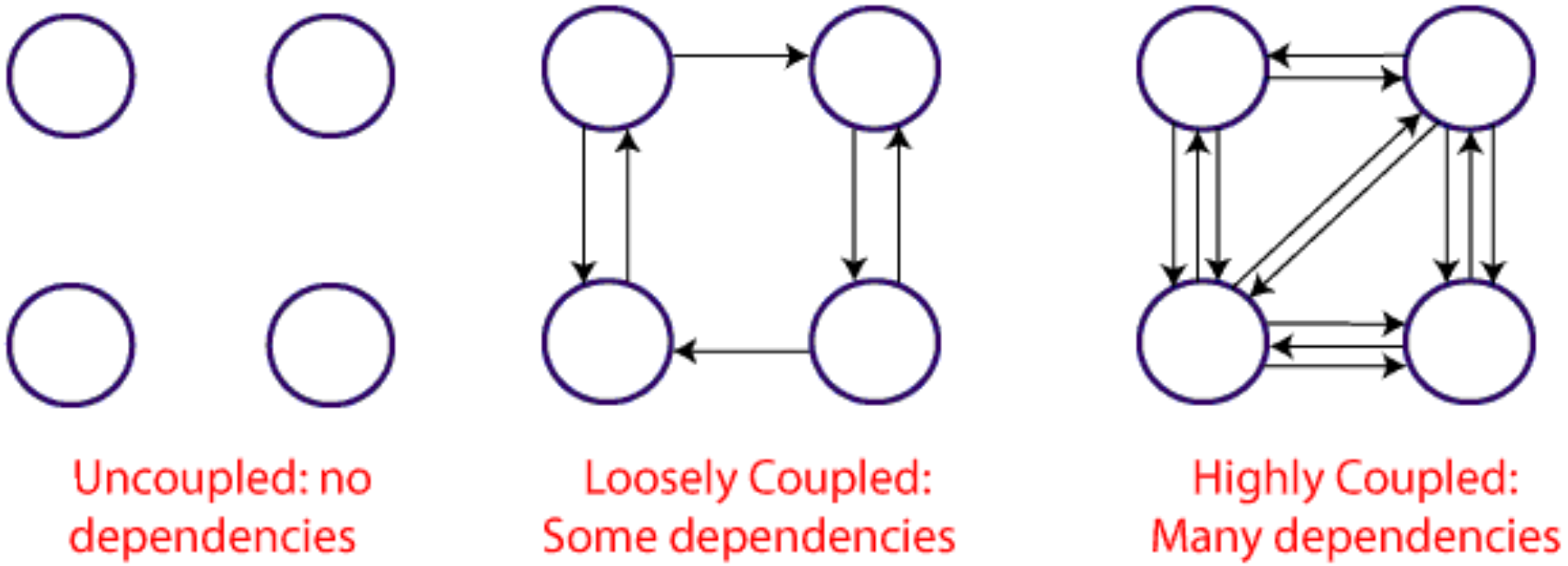
What are signs of poor code?

Circular dependencies: two or more modules depending on each other



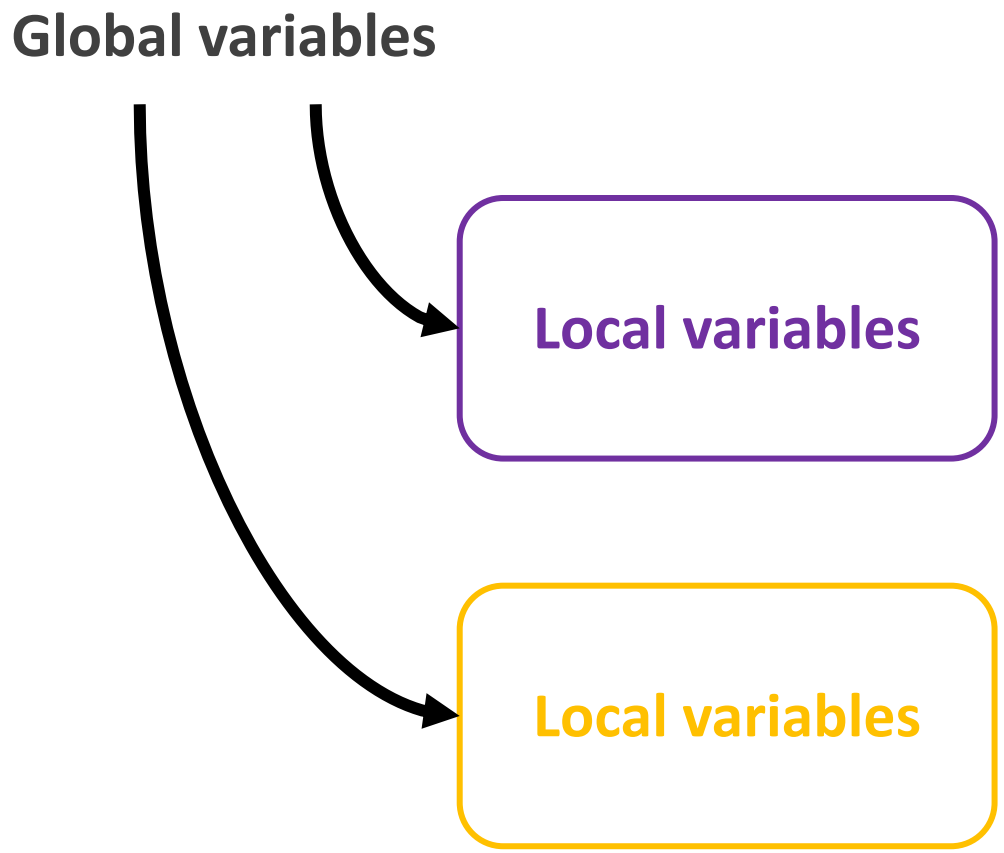
What are signs of poor code?

Hidden coupling: heavily interlinked modules, making one change requires a lot of restructuring



What are signs of poor code?

Overusing global state / context: unnecessary global variables that are modified by multiple agents

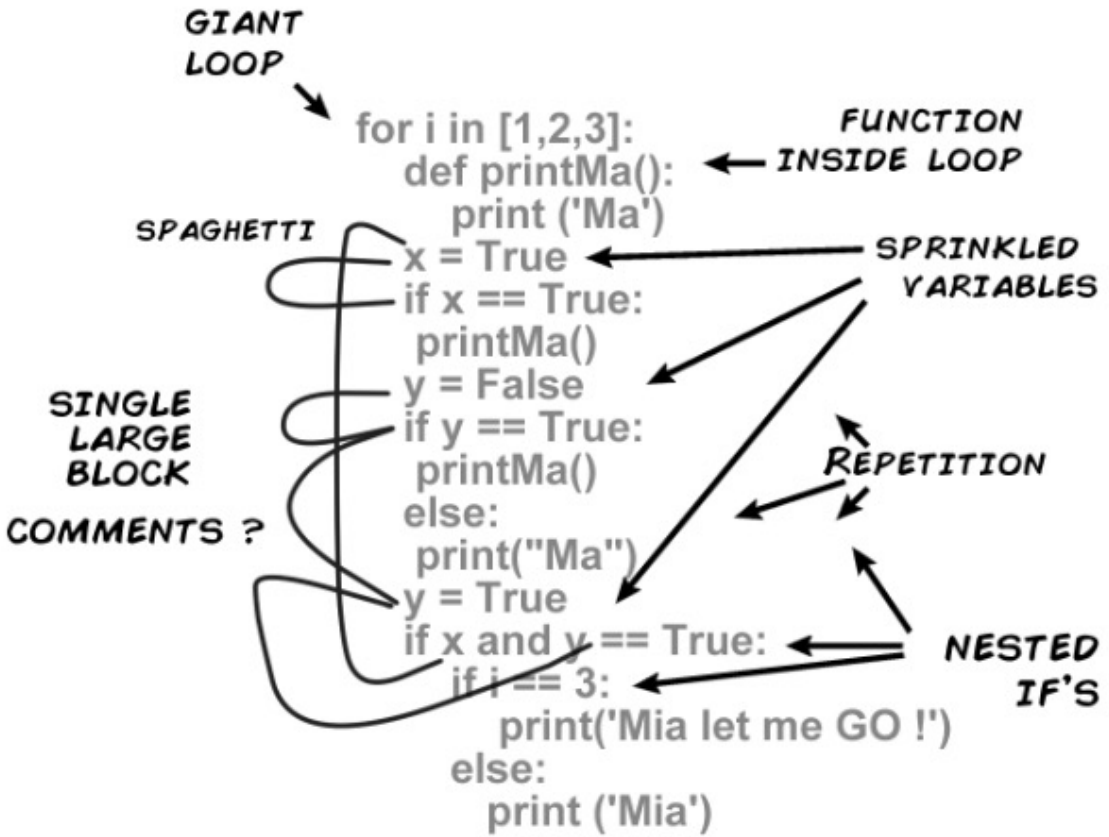


What are signs of poor code?

Spaghetti code: multiple pages of nested if clauses or for loops

Lots of copy-pasted procedural code; no proper segmentation

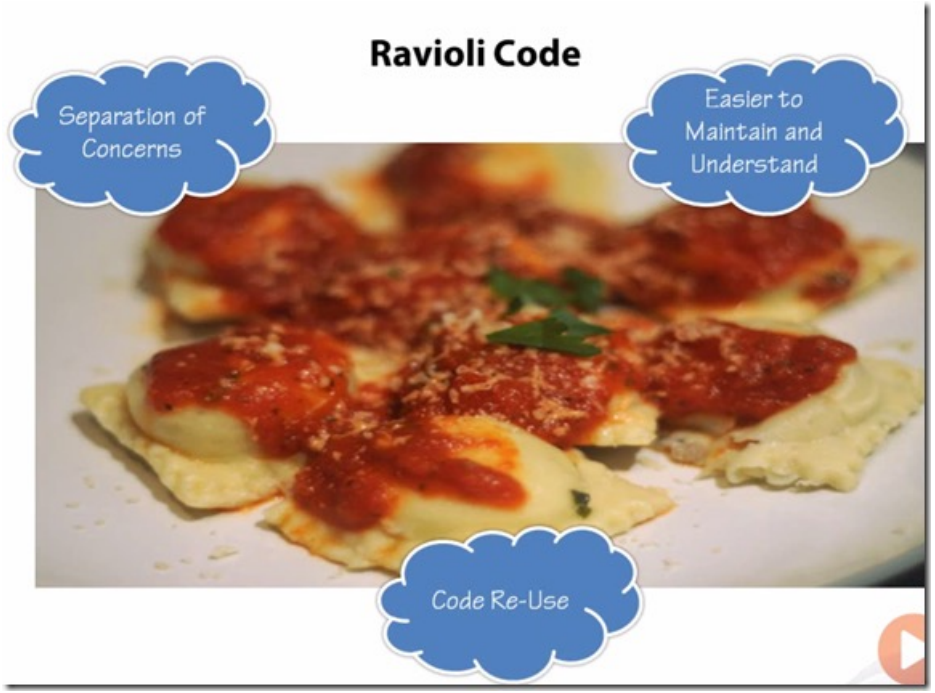
Particularly hard to maintain



Source: https://ichi.pro/de/organisieren-sie-ihren-python-code-222463292355432

What are signs of poor code?

Ravioli code: hundreds of similar pieces of logic (classes or objects) without proper structure;
Hard to remember which module you should use for a given case



Source: https://grekai.wordpress.com/2013/04/03/spa-6-spa-basics-separating-the-ravioli-code-camper/

Summary: What are key signs of poor code?

Circular dependencies: two or more modules depending on each other

Hidden coupling: heavily interlinked modules, making one change requires a lot of restructuring

Overusing global state / context: unnecessary global variables that are modified by multiple agents

Spaghetti code: multiple pages of nested if clauses or for loops

Ravioli code: hundreds of similar pieces of logic (classes or objects) without proper structure; typically you can never remember which module you should use for a given case

Structure of Code in Python

Python naturally encourages structured code (e.g. Modules)

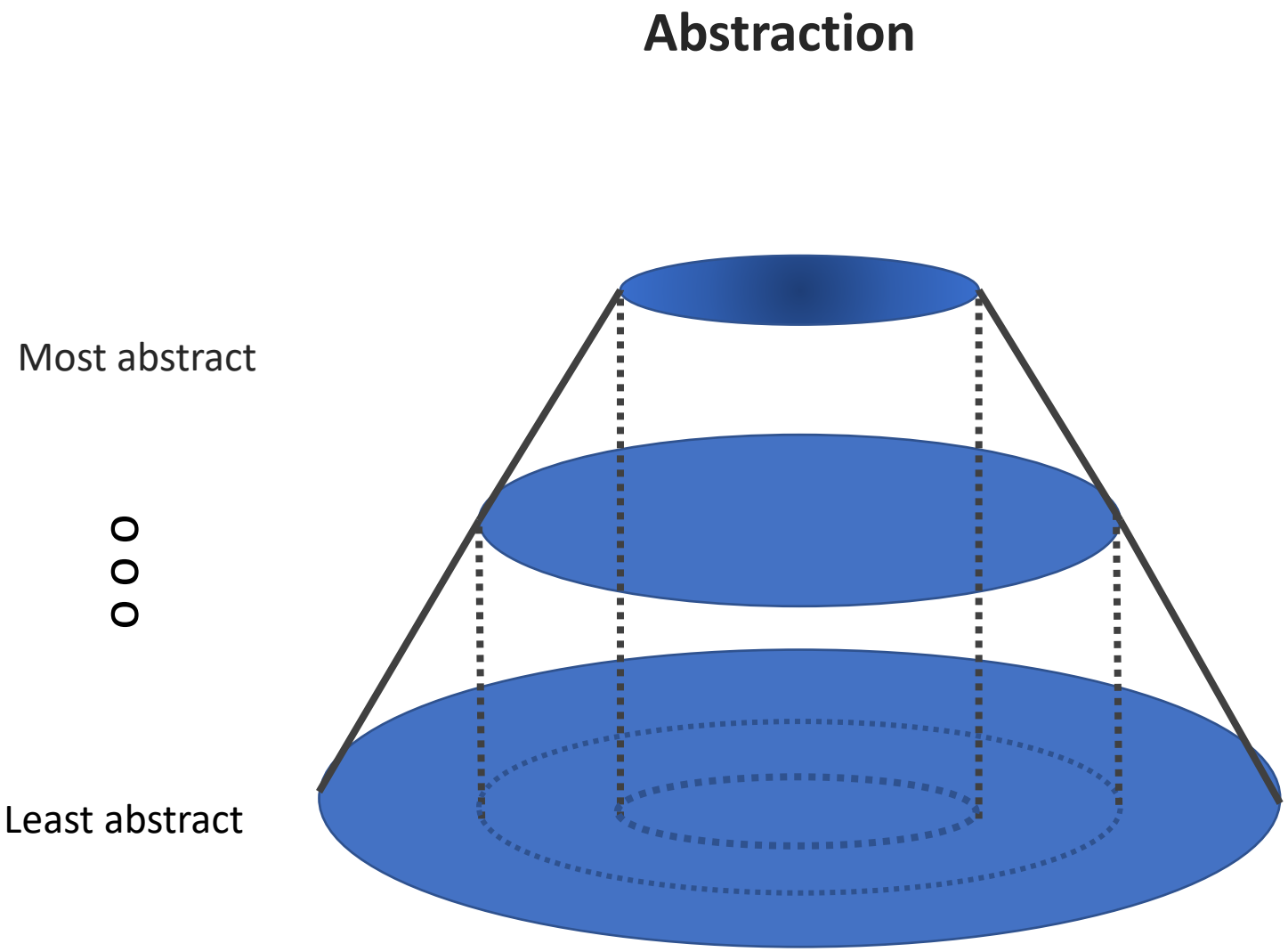
Not many constraints; importing modules is intuitive

Architectural questions: crafting different parts of a project and their interactions

What are signs of poor code?

Which key concepts should we consider for writing efficient code?

Decomposition vs. Abstraction

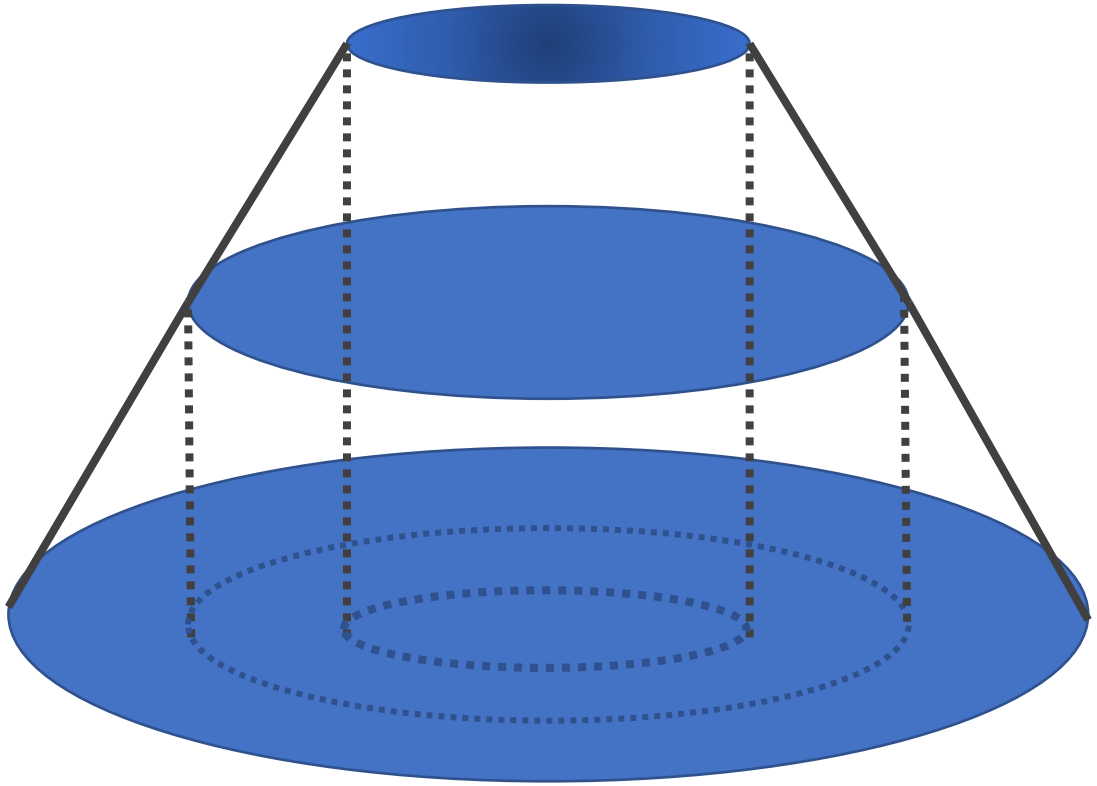


Decomposition vs. Abstraction

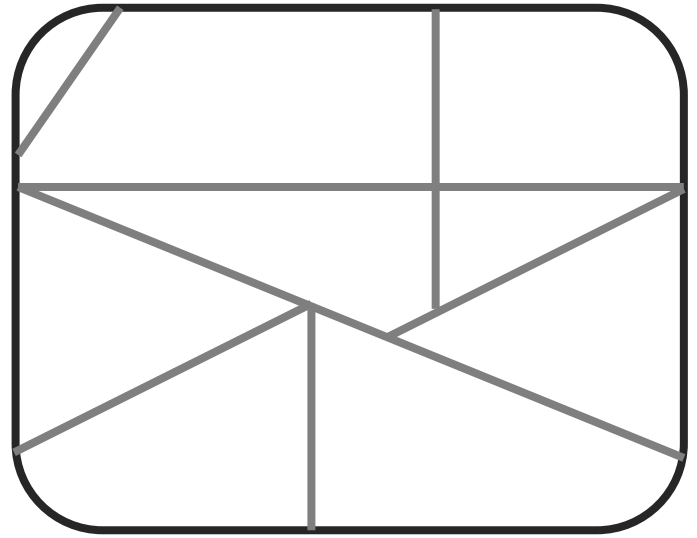
Abstraction

Most abstract

Least abstract



Decomposition



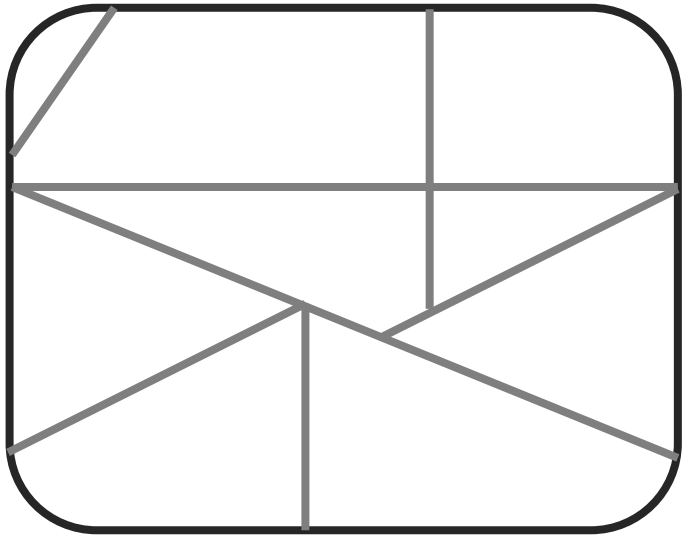
Decomposition “Divide and Conquer”

Tackling large problems with “Divide and Conquer”

Decomposing a problem so that:

- Subproblems with similar level of detail
- Independent solving of subproblems
- Combining solutions to solve the initial problem

Decomposition



Decomposition “Divide and Conquer”

Tackling large problems with “Divide and Conquer”

Decomposing a problem so that:

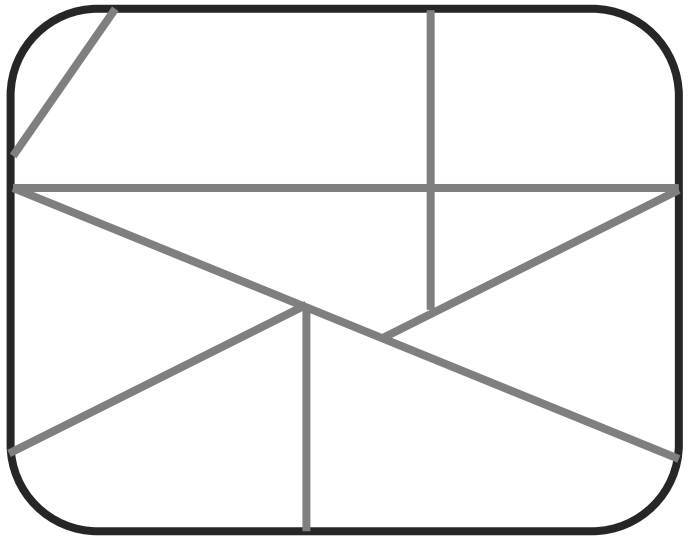
- Subproblems with similar level of detail
- Independent solving of subproblems
- Combining solutions to solve the initial problem

Different devices work together to achieve an end goal

✓ Different people can work on different subproblems
✓ Parallelizing tasks
✓ Maintenance is easier

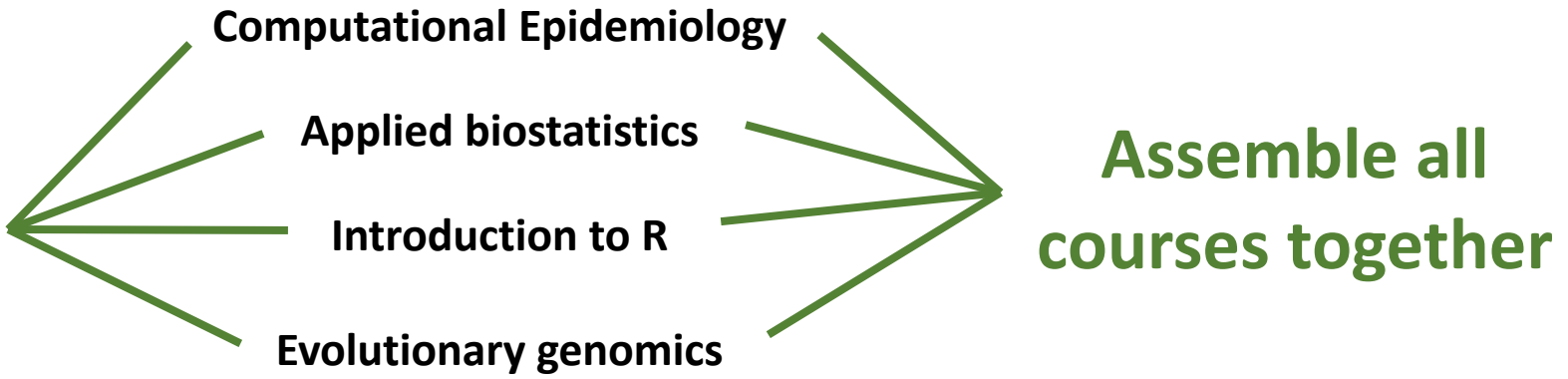
x Solutions to subproblems may not be easy to combine
x Not suitable for poorly understood problems

Decomposition

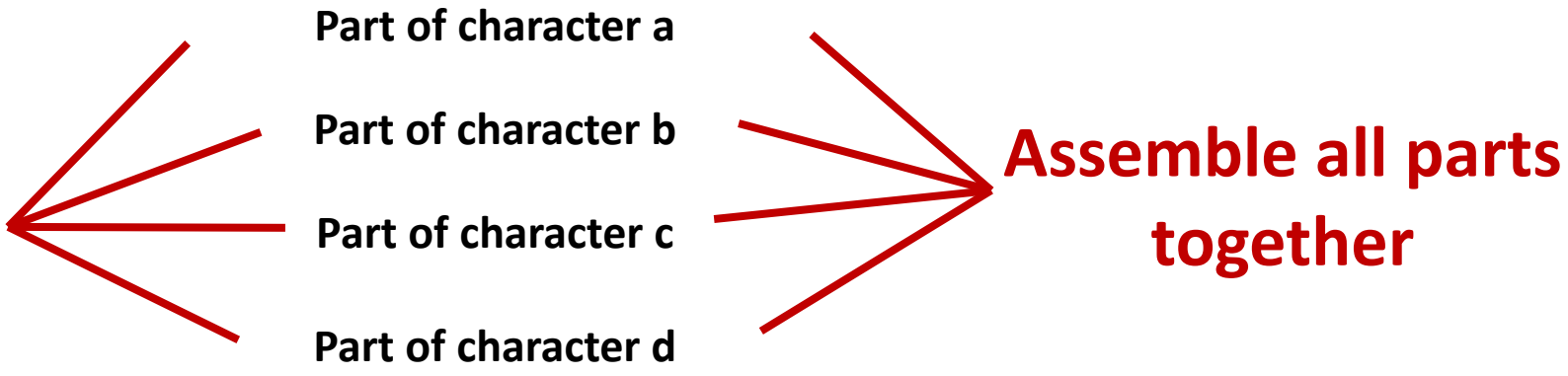


Examples of decomposition

✓ Designing a new study program



x Writing a script

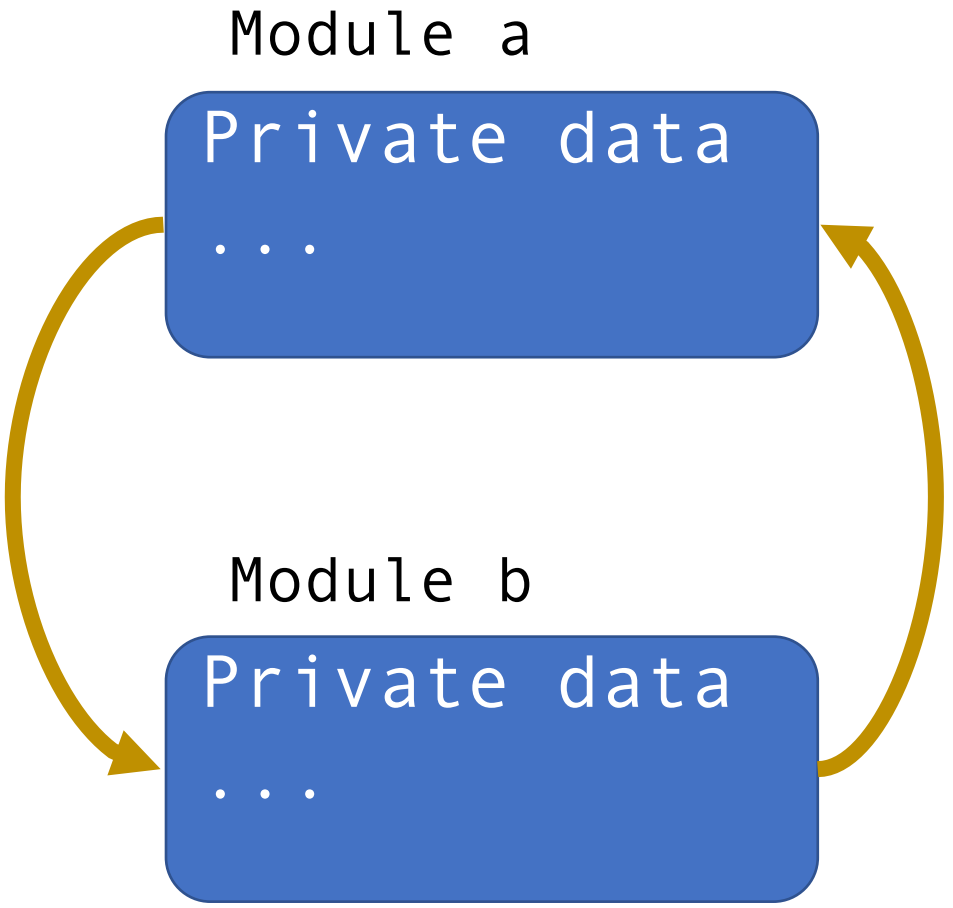


Decomposition

Step 1: What are the components?

- minimize dependencies between components
- coupling across components
- cohesion within each component

- Information hiding
- modules can keep their data private
- limited access procedures

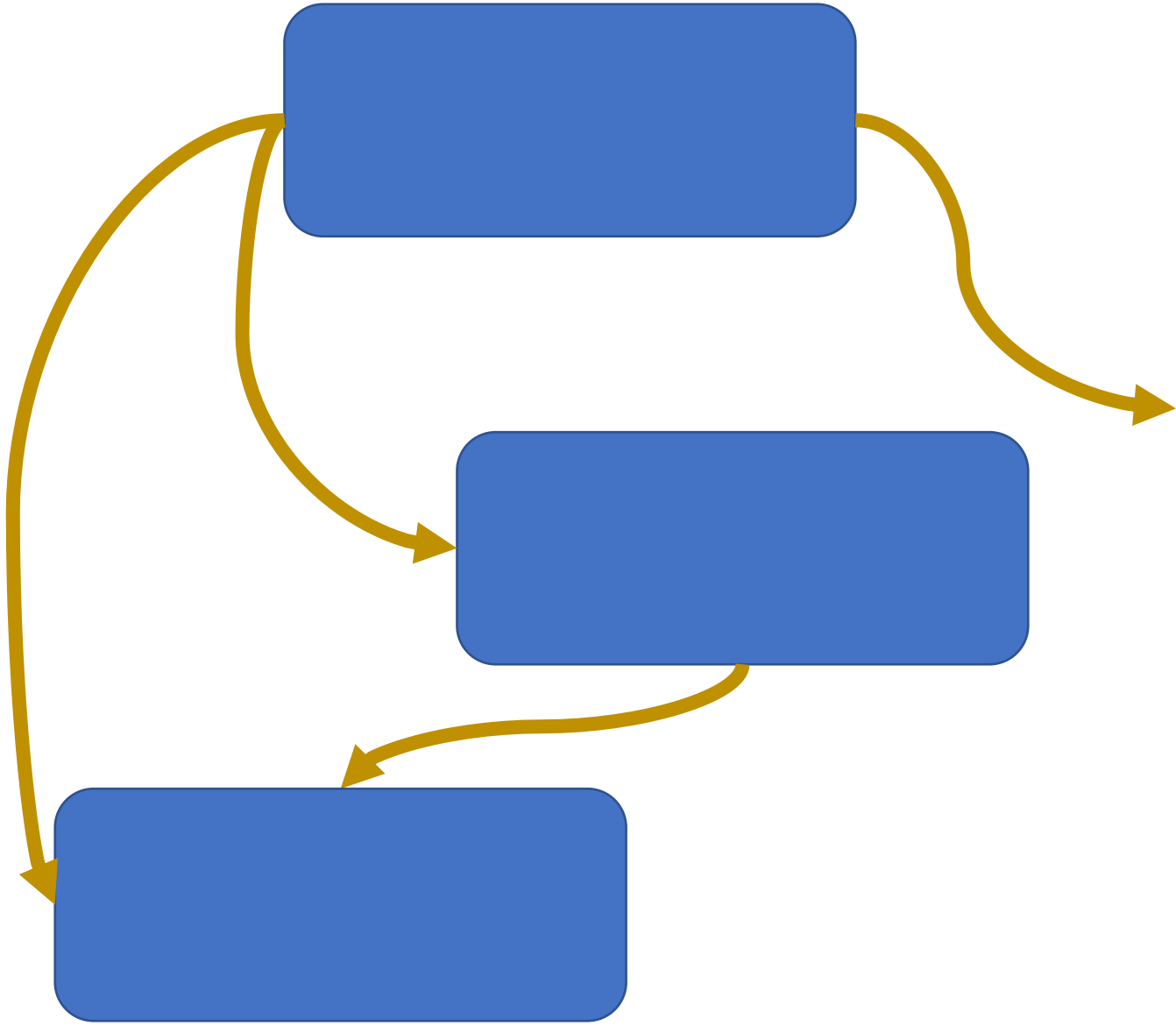


Decomposition

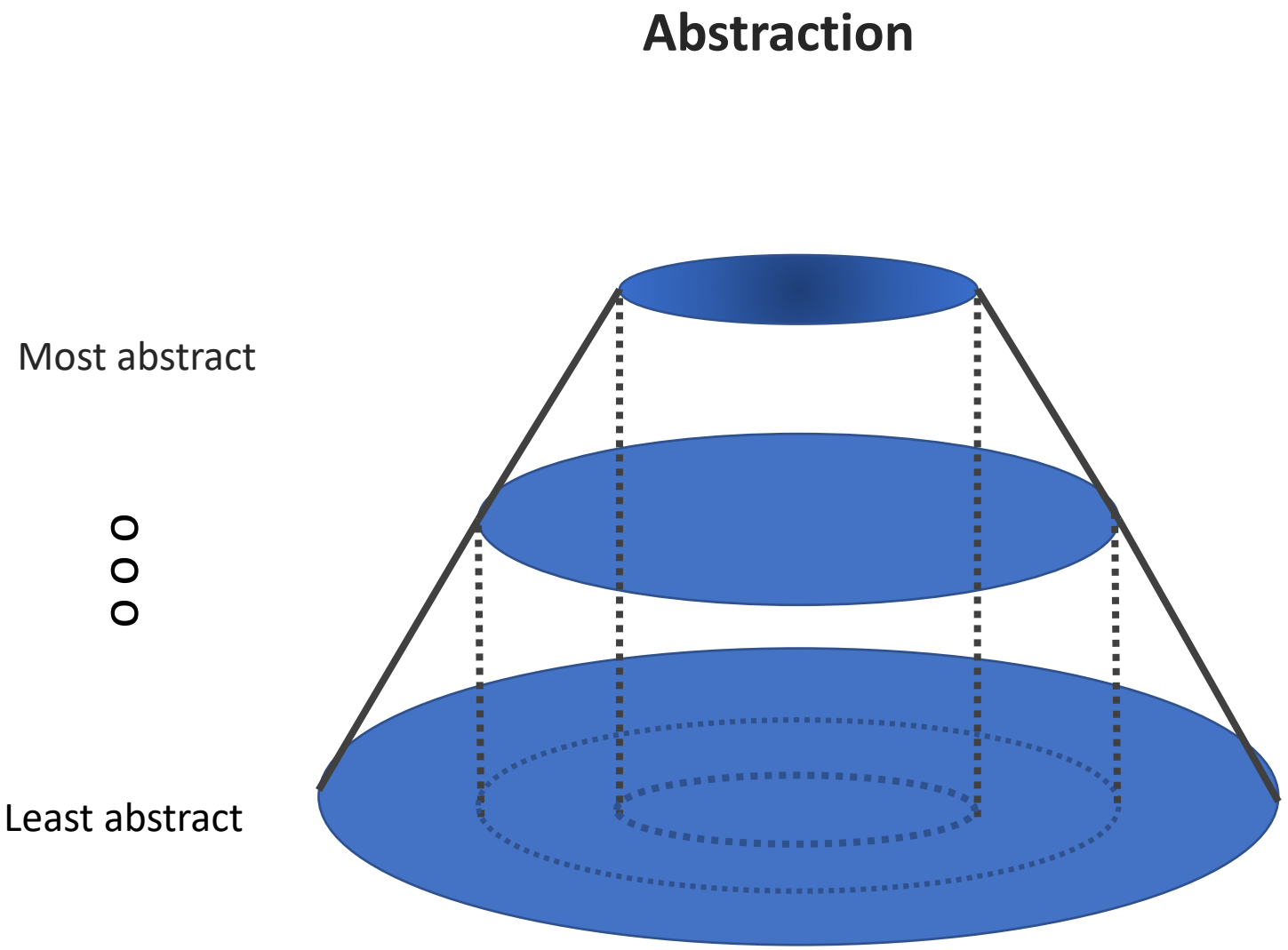
Step 2: Designing the components

- structure
- data flow diagrams
- object diagrams
- …

- coding the components!



Abstraction

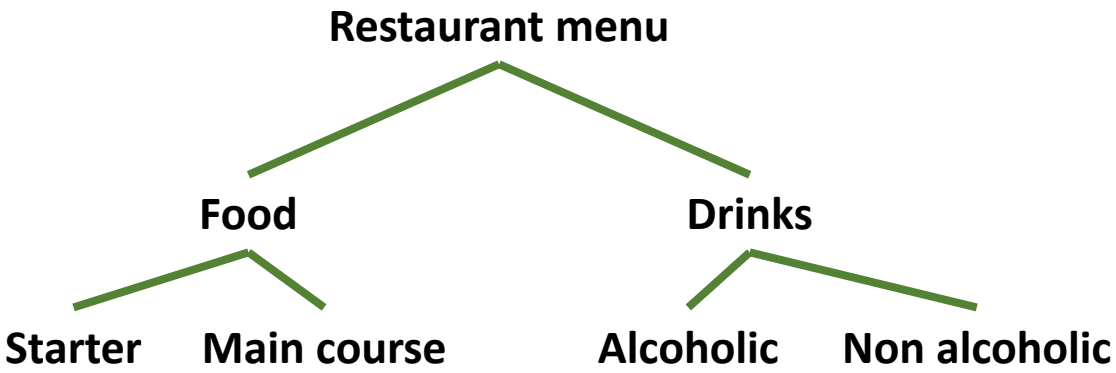


Software reasoning

Abstracting a problem so that:

- Ignore all inconvenient details
- Treat different entities as if they were the same
- Simplify main types of analyses

Abstraction

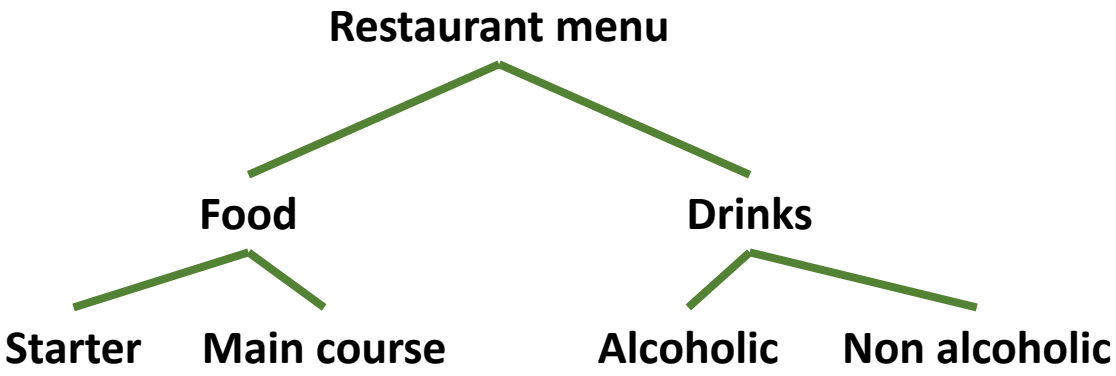


Software reasoning

Abstracting a problem so that:

- Ignore all inconvenient details
- Treat different entities as if they were the same
- Simplify main types of analyses

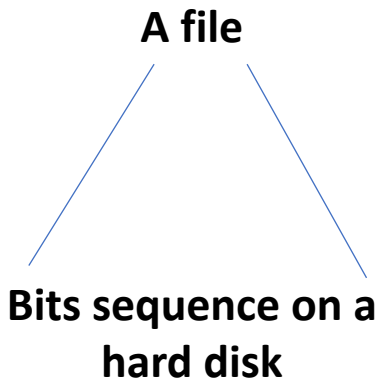
Abstraction



Software reasoning

Abstracting a problem so that:

- Ignore all inconvenient details
- Treat different entities as if they were the same
- Simplify main types of analyses



Abstraction idea: we do not need to know the implementation details to be able to use a file

Abstraction & Decomposition

Abstraction

A piece of code can be seen as a black box

- we do not see details
- we do not need to see the details
- hiding tedious coding details

→ *function specifications or docstrings*

Decomposition

Divide code into modules

- self-contained
- intended to be reusable
- help keep code organized
- keeping code coherent

→*decomposition with functions or classes*

How to create structure with decomposition in Python?

Modules: one of the main decomposition layers in Python

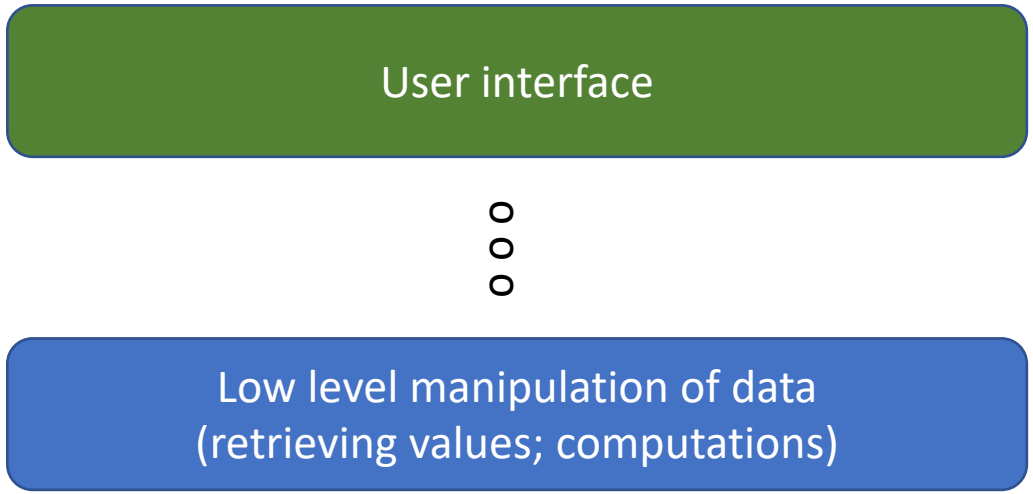
Allows to separate code into parts holding related data and functionality
- self contained / reusable / coherent code / organized code

How to create structure with decomposition in Python?

Modules: one of the main decomposition layers in Python

Allows to separate code into parts holding related data and functionality
- self contained / reusable / coherent code / organized code

Example: One layer of a project can handle interfacing with user actions
Another layer: handles low-level manipulation of data



Structuring code with modules

Modules can be:

- built-in (os / sys)
- Third party modules that are installed in environment
- Internal modules written for a project

Modules can be named:

- Short
- Lowercase
- Avoiding special symbols (. ?)

Python expects file ”module.py” inside folder “my”

Example of what to avoid: `my.module.py`

Handling modules into our code

How to best import modules?

`import mymodule` will :

(a) search for the file mymodule.py in our working directory;
(b) if it does not exist, it will search for mymodule.py in the python path recursively;
(c) if it does not exist an `ImportError` exception will be raised

What are practices for importing modules?

Importing modules

Showing what is imported in the local namespace is advised

`import *` will not show explicitly what is imported

Very bad

```
[...]
from modu import *
[...]
x = sqrt(4)  # Is sqrt part of modu? A builtin? Defined above?
```

Better

```
from modu import sqrt
[...]
x = sqrt(4)  # sqrt may be part of modu, if not redefined in between
```

Best

```
import modu
[...]
x = modu.sqrt(4)  # sqrt is visibly part of modu's namespace
```

Decomposition with functions

Functions: reusable pieces of code
They only run when 'called'

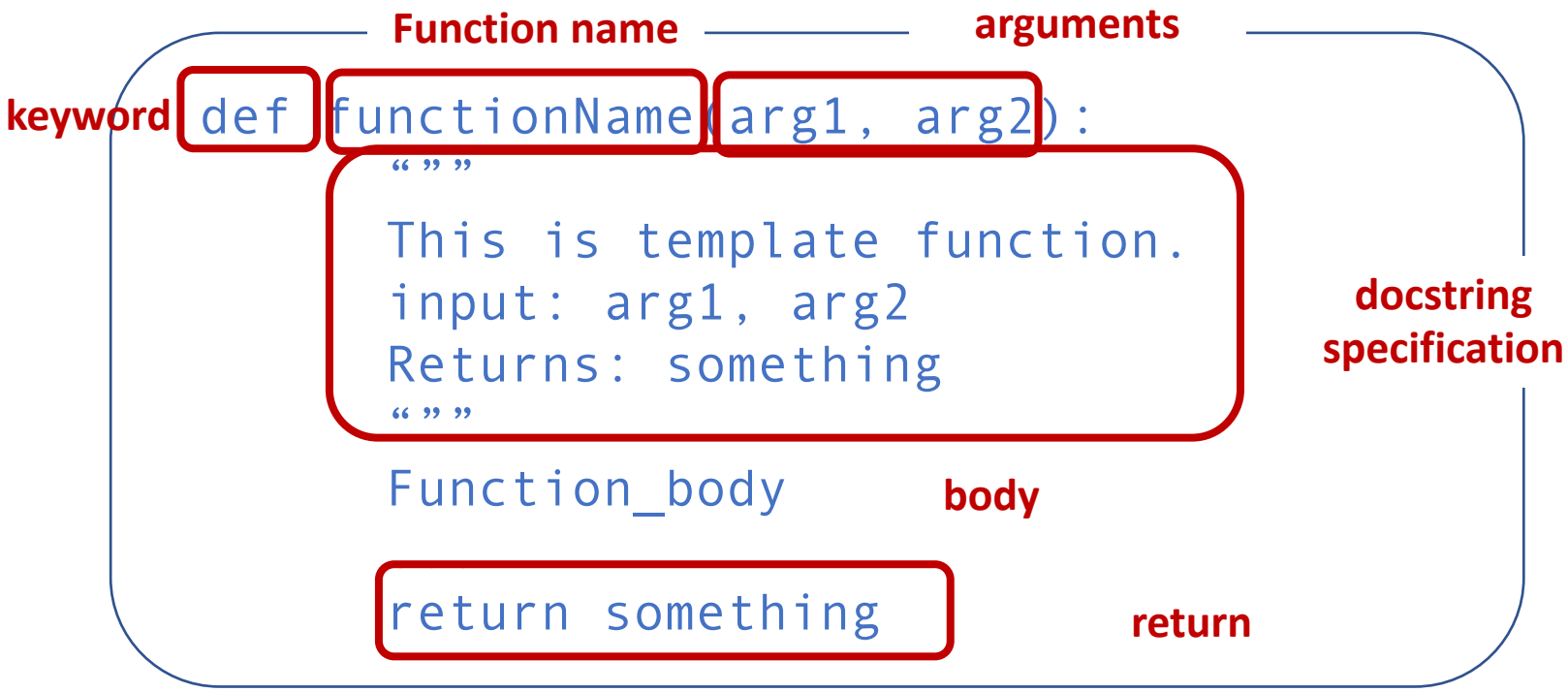
```
def functionName(arg1, arg2):
        """
        This is template function.
        input: arg1, arg2
        Returns: something
        """
        Function_body
        ...
        return something
```

```
def subt(num1, num2):
        """ subtraction function """
        return num1-num2

>> subt(10,2)
>> 8
```

Decomposition with functions

Functions: reusable pieces of code
They only run when 'called'




```
def subt(num1, num2):
        """ subtraction function """
        return num1-num2

>> subt(10,2)
>> 8
```

Function types

- Built-in functions : e.g. `min() type()`

- User-defined functions

- Anonymous functions (`lambda`)

Decomposition with Methods (reminder)

```
class ClassName:

      def MethodName():
      …….
      Method_body
      ……..


class Furniture(object):
      def my_method(self):
            print(“This is a chair”)

chair = Furniture()
chair.my_method()

This is a chair
```

Functions vs. Methods

A. Methods are called on an object

We call our method: “my_method” on the object “chair”

A method can access that data within the object

Functions are called without any object

B. Methods can alter the object’s state

Functions operate on an object

Methods: Functions which belong to an object

Reminder: object-oriented programming in python

10. Classes and Object-Oriented Programming

Topics

- Procedural and Object-Oriented Programming
- Classes
- Working with Instances
- Techniques for Designing Classes

11. Inheritance

Topics

- Introduction to Inheritance
- Polymorphism

Please refer to: Basic programming for non-informaticians. With practicals.

Outline for today

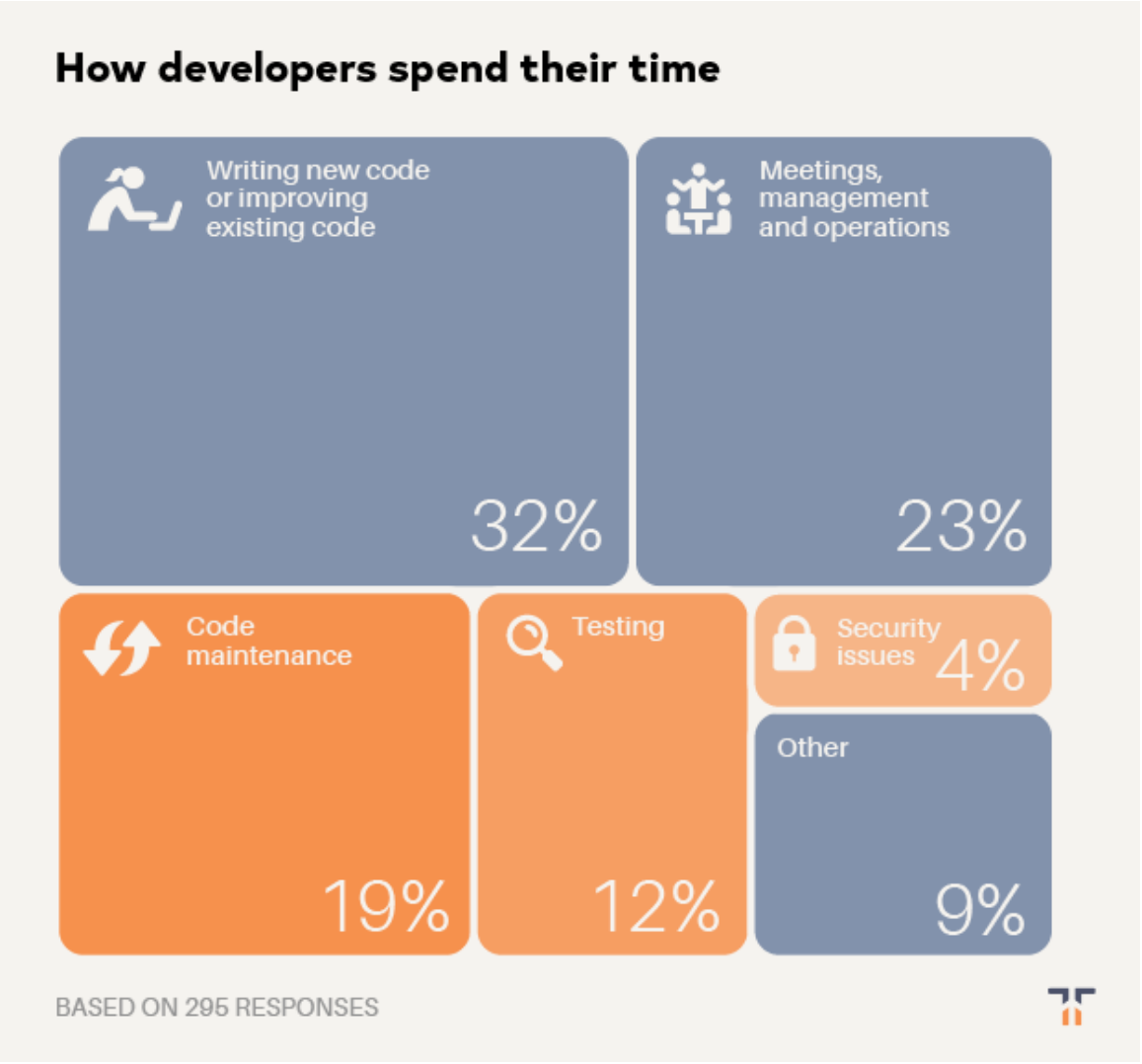
1. Coding principles for writing efficient code
2. **Coding styles**
3. Source Code Analysis

“Code is read more often than it is written”

Guido van Rossum

“Code is read more often than it is written”

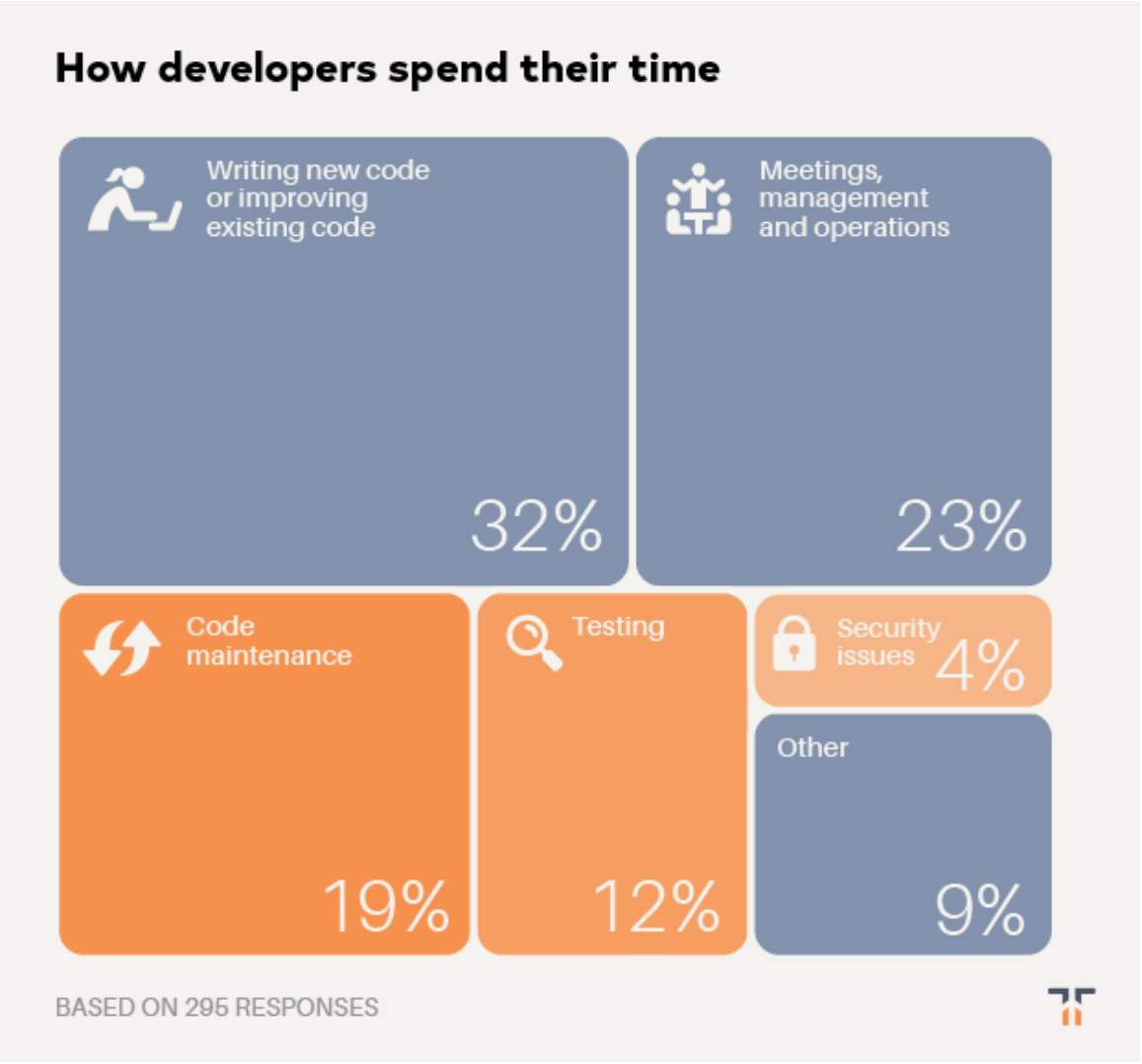
Guido van Rossum



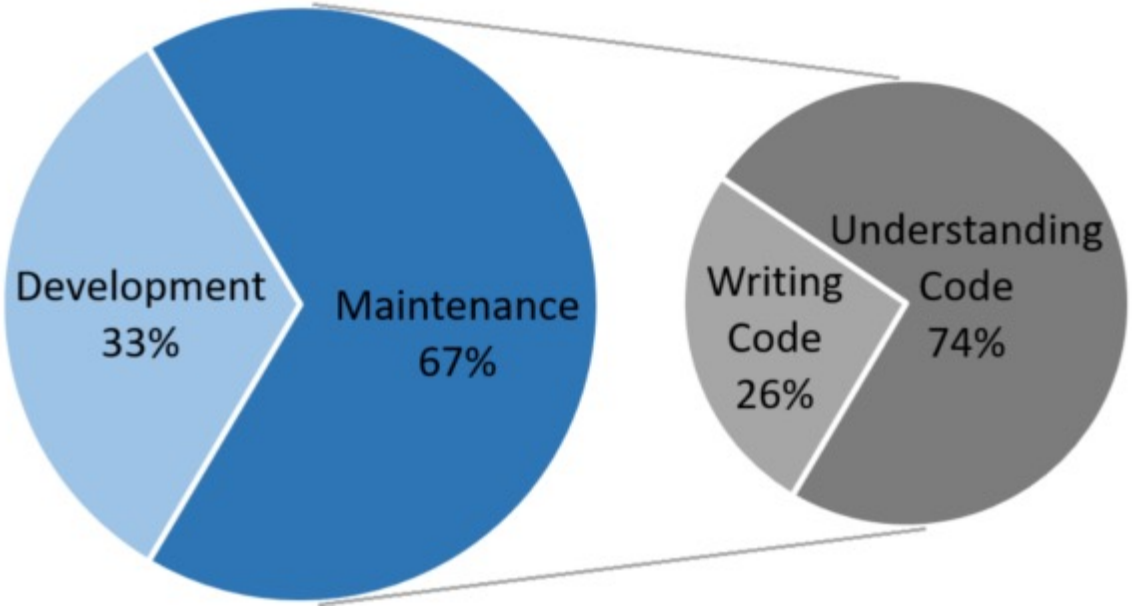
Source: https://thenewstack.io/

“Code is read more often than it is written”

Guido van Rossum



Source: https://thenewstack.io/



Source: https://adadevelopment.github.io/engineering/code-as-documentation.html

Python styles and coding guidelines

- ✓ Consistent code
- ✓ Easy to read and understand
- ✓ Facilitates collaborative work

"Code is read more often than it is written"

Guido van Rossum

PEP 8 -- Style Guide for Python Code

Python Enhancement Proposals

https://www.python.org/dev/peps/pep-0008/

Reminder: The ZEN of Python

```
import this
```

```
The Zen of Python, by Tim Peters

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!
```

Python Enhancement Proposal

PEP 8 -- Style Guide for Python Code

- ✓ Design and style
- ✓ *Improves code readability*
- ✓ *Facilitates collaboration with others*

What does it cover?

- *Naming of variables*
- *Whitespaces*
- *Commenting of code*

==> More readable code; easier to understand & maintain

Naming Conventions

```
import this
```

```
The Zen of Python, by Tim Peters

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!
```

Naming Conventions

Naming for:

- Variables
- Functions
- Classes
- Packages…

Can we figure out from the name what it represents?

Type	Naming example	
Function	my_function , function	
Variable	x, var, my_var	
Class	MyClass, Model	← *No underscores*
Method	method, my_method	← *Similar to ”Variable” (without single letter)*
Constant	CONSTANT, MY_CONSTANT	
Module	my_module.py, module.py	
Package	package, mypackage	← *No underscores*

Naming Conventions: How to choose names?

Descriptive names make clear what an object represents
Example:

Variables

```
# Not recommended
a = 'Python, Bioinformatics'
b, c = a.split()

print(b)
print(c)
```

```
Python,
Bioinformatics
```

```
# Recommended
course_program = 'Python, Bioinformatics'
course, program = course_program.split()

print(course)
print(program)
```

```
Python,
Bioinformatics
```

Naming Conventions: How to choose names?

Descriptive names make clear what an object represents
Example:

Variables

```
# Not recommended
a = 'Python, Bioinformatics'
b, c = a.split()

print(b)
print(c)
```

```
Python,
Bioinformatics
```

```
# Recommended
course_program = 'Python, Bioinformatics'
course, program = course_program.split()

print(course)
print(program)
```

```
Python,
Bioinformatics
```

Functions

```
# Not recommended

def mp(a):
    return a * 100

mp(2)
```

```
200
```

```
# Recommended

def myltiply_by_hundred(a):
    return a * 100

myltiply_by_hundred(2)
```

```
200
```

Code Layout

```
import this
```

```
The Zen of Python, by Tim Peters

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!
```

Code Layout

Laying out code can affect readability
Too many blank lines: sparse looking code; too few: unreadable

- Code Lay-out
  - Indentation
  - Tabs or Spaces?
  - Maximum Line Length
  - Should a Line Break Before or After a Binary Operator?
  - Blank Lines
  - Source File Encoding
  - Imports
  - Module Level Dunder Names

Use 4 spaces per indentation level

Spaces are the preferred indentation method*

Limit all lines to a maximum of 79 characters

Break lines before binary operators

Surround top-level function and class definitions with two blank lines.

Imports should usually be on separate line

** According to PEP8. At least do not mix spaces with tabs*

Code Layout

```
import this
```

```
The Zen of Python, by Tim Peters

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!
```

Emphasizing simplicity in code

Often: several ways to perform a similar action
How to simplify?

1. Not comparing Boolean values to True or False

```
# Not recommended

compar = 100 > 0
if compar == True:
    print('True, 100 is greater than 0')
```

```
True, 100 is greater than 0
```

Equivalent ways to code the same thing

```
# Recommended

compar = 100 > 0
if compar:
    print('True, 100 is greater than 0')
```

```
True, 100 is greater than 0
```

Simpler one is preferred

Emphasizing simplicity in code

Often: several ways to perform a similar action
How to simplify?

2. Use the fact that empty sequences are false:

```
# Not recommended

my_list = []
if not len(my_list):
    print('Empty list')
```

```
Empty list
```

We may be tempted to compute the length of a list (empty lists have a length of 0)

```
# Recommended

my_list = []
if not my_list:
    print('Empty list')
```

```
Empty list
```

Empty lists in python are False by default!

Emphasizing simplicity in code

Often: several ways to perform a similar action
How to simplify?

3. Use ''.startswith() and ''.endswith() instead of string slicing to check for prefixes or suffixes.

```
# Not recommended

if foo[:3] == 'bar':
    print('Starting with bar')
```

We may be tempted to use slicing
Prone to errors!

```
# Recommended

if foo.startswith('bar'):
    print('Starting with bar')
```

Less prone to errors & easier to read and understand!

Emphasizing simplicity in code

Often: several ways to perform a similar action
How to simplify?

1. Not comparing Boolean values to True or False

2. Use the fact that empty sequences are false:

3. Use ''.startswith() and ''.endswith() instead of string slicing to check for prefixes or suffixes.

And many more:

https://www.python.org/dev/peps/pep-0008/#code-lay-out

Should we ignore PEP 8?

Generally: it is recommended to follow PEP 8

Clean & Professional & Readable code

However, we may need to ignore PEP 8:

- if this breaks compatibility with existing code / software coding standards

- If code has to be compatible with older Python versions

Examples of code

Good or bad examples?

```
my_function({variable: 3}, my_list[2], [])

my_function( { variable: 3 }, my_list[ 2 ], )
```

```
i > 100

i==100
```

Examples of code

Good or bad examples?

```
my_function({variable: 3}, my_list[2], [])

my_function( { variable: 3 }, my_list[ 2 ], )
```

x Bad: Whitespace inside parentheses, brackets, or braces

```
i > 100

i==100
```

✓ **Good: Operators are surrounded with a single whitespace on either side**

Examples of code

Good or bad examples?

```
if attr:
    print('attr is true!')

if not attr:
    print('attr is false!')
```

```
if attr == True:
    print('True!')

if attr == None:
    print('attr is None!')
```

Examples of code

Good or bad examples?

```
if attr:
    print('attr is true!')

if not attr:
    print('attr is false!')
```

✓ **Good: simply checking the value of our attribute**

x Bad

```
if attr == True:
    print('True!')

if attr == None:
    print('attr is None!')
```

Examples of code

Good or bad examples?

```
my_text = """For a long time I used to go to bed early. Sometimes, \
    when I had put out my candle, my eyes would close so quickly that I had not even \
    time to say “I’m going to sleep.”"""

from some.deep.module.inside.a.module import a_nice_function, another_nice_function, \
    yet_another_nice_function
```

```
my_text = (
    "For a long time I used to go to bed early. Sometimes, "
    "when I had put out my candle, my eyes would close so quickly "
    "that I had not even time to say “I’m going to sleep.”"
)

from some.deep.module.inside.a.module import (
    a_nice_function, another_nice_function, yet_another_nice_function)
```

Examples of code

Good or bad examples?

x Bad

```
my_text = """For a long time I used to go to bed early. Sometimes, \
    when I had put out my candle, my eyes would close so quickly that I had not even \
    time to say “I’m going to sleep.”"""

from some.deep.module.inside.a.module import a_nice_function, another_nice_function, \
    yet_another_nice_function
```

✓ **Good: avoiding backslashes; using parentheses**

```
my_text = (
    "For a long time I used to go to bed early. Sometimes, "
    "when I had put out my candle, my eyes would close so quickly "
    "that I had not even time to say “I’m going to sleep.”"
)

from some.deep.module.inside.a.module import (
    a_nice_function, another_nice_function, yet_another_nice_function)
```

How can we control the quality of our code?

Linters

Programs that analyse flag errors

Suggestions for how to fix the error

You can add them as extensions to your text editors

Formatters

Programs that refactor your code to conform with PEP 8

Outline for today

1. Coding principles for writing efficient code
2. Coding styles
3. **Source Code Analysis**

Static analysis of code

- Analyze compiled code or source code
- Help detect vulnerabilities without executing the code
- Execute quickly compared to dynamic analysis
- Towards automating code quality maintenance
- Help with refactoring (detecting duplicate / unused code)

Structuring python code

Exemplar python file

python_file.py

```
1  class furniture:
2      def _init_(self, room):
3          self.room = room
4
5  my_chair = char('kitchen')
6
7  def   pricing(furniture, store)  :
8      furniture.price = 230
9
10 pricing(chair('kitchen'), store1)
11
```

Do you see anything problematic with this file?

Structuring python code

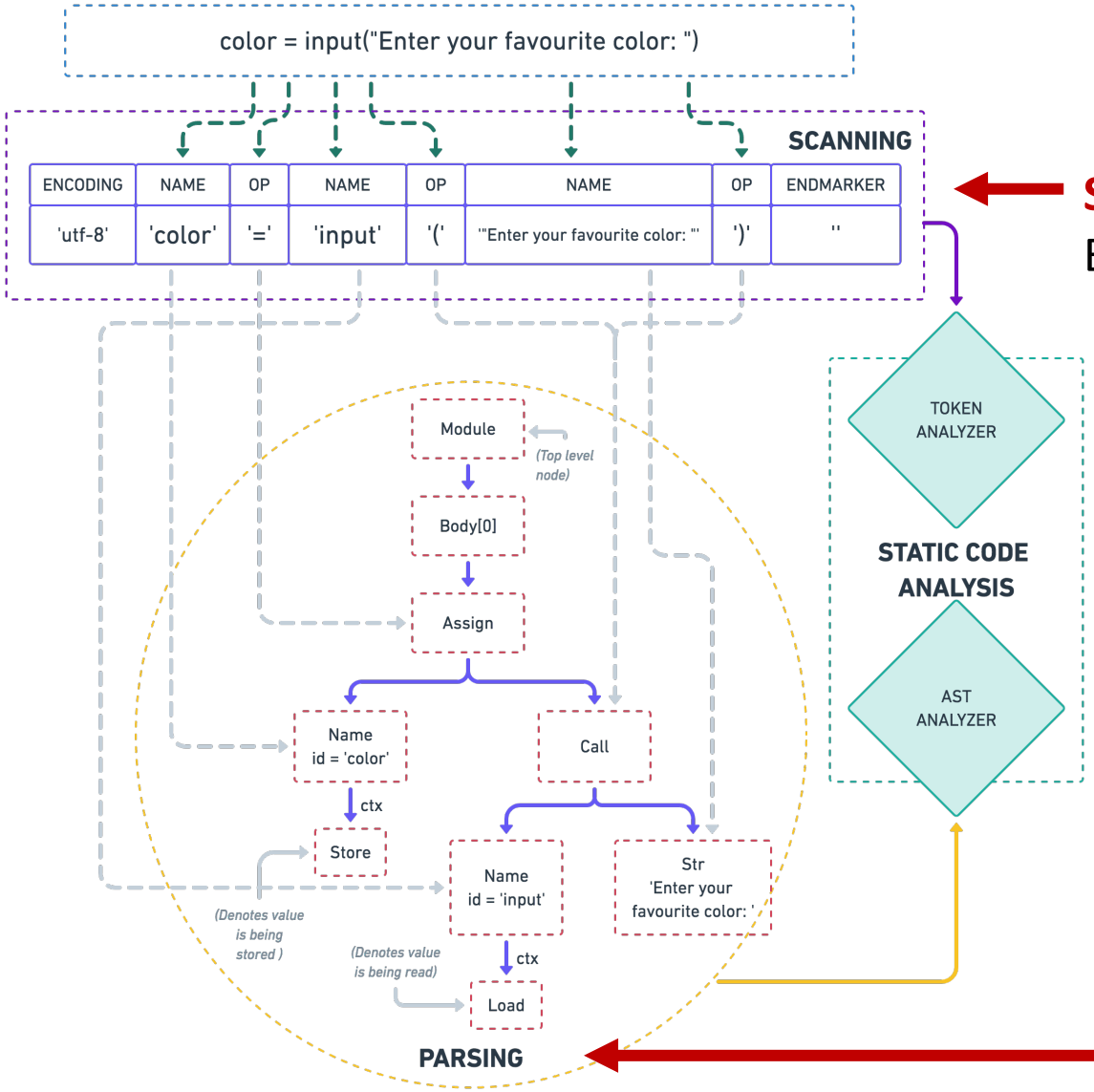
Exemplar python file

python_file.py

```
1  class furniture:
2      def _init_(self, room):
3          self.room = room
4
5  my_chair = char('kitchen')
6
7  def   pricing(furniture, store)  :
8      furniture.price = 230
9
10 pricing(chair('kitchen'), store1)
11
```

Do you see anything problematic with this file?
How can we detect problems automatically?

Decomposing code



Scanning:
Breaking down code into smaller chunks "*tokens*"

Parser:
- validates that the sequence of tokens conforms to python's grammar
- organizes them in a tree-like structure
- high-level structure of the program.

Abstract Syntax Tree -AST-

https://deepsource.io/blog/introduction-static-code-analysis/

Structuring python code

```
pylint python_file.py
```

```
staff-205-62:Structuring athina$ pylint python_file.py
No config file found, using default configuration
************* Module python_file
C:  7, 0: No space allowed before :
def   pricing(furniture, store)  :
                                 ^ (bad-whitespace)
C:  1, 0: Missing module docstring (missing-docstring)
C:  1, 0: Class name "furniture" doesn't conform to PascalCase naming style (invalid-name)
C:  1, 0: Missing class docstring (missing-docstring)
W:  3, 8: Attribute 'room' defined outside __init__ (attribute-defined-outside-init)
R:  1, 0: Too few public methods (0/2) (too-few-public-methods)
C:  5, 0: Constant name "my_chair" doesn't conform to UPPER_CASE naming style (invalid-name)
E:  5,11: Undefined variable 'char' (undefined-variable)
W:  7,14: Redefining name 'furniture' from outer scope (line 1) (redefined-outer-name)
C:  7, 0: Missing function docstring (missing-docstring)
W:  7,25: Unused argument 'store' (unused-argument)
E: 10, 8: Undefined variable 'chair' (undefined-variable)
E: 10,26: Undefined variable 'store1' (undefined-variable)

-----------------------------------
Your code has been rated at -25.71/10
```

Structuring python code

```
pylint python_file.py
```

```
staff-205-62:Structuring athina$ pylint python_file.py
No config file found, using default configuration
************* Module python_file
C:  7, 0: No space allowed before :
def   pricing(furniture, store)  :
                                 ^ (bad-whitespace)
C:  1, 0: Missing module docstring (missing-docstring)
C:  1, 0: Class name "furniture" doesn't conform to PascalCase naming style (invalid-name)
C:  1, 0: Missing class docstring (missing-docstring)
W:  3, 8: Attribute 'room' defined outside __init__ (attribute-defined-outside-init)
R:  1, 0: Too few public methods (0/2) (too-few-public-methods)
C:  5, 0: Constant name "my_chair" doesn't conform to UPPER_CASE naming style (invalid-name)
E:  5,11: Undefined variable 'char' (undefined-variable)
W:  7,14: Redefining name 'furniture' from outer scope (line 1) (redefined-outer-name)
C:  7, 0: Missing function docstring (missing-docstring)
W:  7,25: Unused argument 'store' (unused-argument)
E: 10, 8: Undefined variable 'chair' (undefined-variable)
E: 10,26: Undefined variable 'store1' (undefined-variable)

-----------------------------------
Your code has been rated at -25.71/10
```

C: coding style

W: warning

E: Errors

Line in file

“Score” for our code

How can we fix our code?

Structuring python code

```
pylint python_file.py
```

```
staff-205-62:Structuring athina$ pylint python_file.py
No config file found, using default configuration
************* Module python_file
C:  7, 0: No space allowed before :
def   pricing(furniture, store)  :
                                 ^ (bad-whitespace)
C:  1, 0: Missing module docstring (missing-docstring)
C:  1, 0: Class name "furniture" doesn't conform to PascalCase naming style (invalid-name)
C:  1, 0: Missing class docstring (missing-docstring)
W:  3, 8: Attribute 'room' defined outside __init__ (attribute-defined-outside-init)
R:  1, 0: Too few public methods (0/2) (too-few-public-methods)
C:  5, 0: Constant name "my_chair" doesn't conform to UPPER_CASE naming style (invalid-name)
E:  5,11: Undefined variable 'char' (undefined-variable)
W:  7,14: Redefining name 'furniture' from outer scope (line 1) (redefined-outer-name)
C:  7, 0: Missing function docstring (missing-docstring)
W:  7,25: Unused argument 'store' (unused-argument)
E: 10, 8: Undefined variable 'chair' (undefined-variable)
E: 10,26: Undefined variable 'store1' (undefined-variable)

-----------------------------------
Your code has been rated at -25.71/10
```

C: coding style

W: warning

E: Errors

Line in file

“Score” for our code

How can we fix our code?

Improving our python file

Earlier version

python_file.py

```
1  class furniture:
2      def _init_(self, room):
3          self.room = room
4
5  my_chair = char('kitchen')
6
7  def   pricing(furniture, store)  :
8      furniture.price = 230
9
10 pricing(chair('kitchen'), store1)
```

Extra space

Newer version

python_file.py

```
1  class furniture:
2      def _init_(self, room):
3          self.room = room
4
5  my_chair = char('kitchen')
6
7  def   pricing(furniture, store):
8      furniture.price = 230
9
10 pricing(chair('kitchen'), store1)
```

Checking code again

```
pylint python_file.py
```

```
--------------------------------------------------------------------
Your code has been rated at -24.29/10 (previous run: -25.71/10, +1.43)

staff-205-62:Structuring athina$ pylint python_file.py
No config file found, using default configuration
************* Module python_file
C:  5, 0: Missing class docstring (missing-docstring)
W:  7, 8: Attribute 'room' defined outside __init__ (attribute-defined-outside-init)
R:  5, 0: Too few public methods (0/2) (too-few-public-methods)
C:  9, 0: Constant name "my_chair" doesn't conform to UPPER_CASE naming style (invalid-name)
C: 11, 0: Missing function docstring (missing-docstring)
W: 11,25: Unused argument 'store' (unused-argument)
E: 14, 8: Undefined variable 'chair' (undefined-variable)
E: 14,26: Undefined variable 'store1' (undefined-variable)

--------------------------------------------------------------------
Your code has been rated at -12.86/10 (previous run: -24.29/10, +11.43)

staff-205-62:Structuring athina$ pylint python_file.py
No config file found, using default configuration
************* Module python_file
W:  8, 8: Attribute 'room' defined outside __init__ (attribute-defined-outside-init)
R:  5, 0: Too few public methods (0/2) (too-few-public-methods)
C: 10, 0: Constant name "my_chair" doesn't conform to UPPER_CASE naming style (invalid-name)
C: 12, 0: Missing function docstring (missing-docstring)
W: 12,25: Unused argument 'store' (unused-argument)
E: 15, 8: Undefined variable 'chair' (undefined-variable)
E: 15,26: Undefined variable 'store1' (undefined-variable)

--------------------------------------------------------------------
Your code has been rated at -11.43/10 (previous run: -12.86/10, +1.43)
```

There are still a few things to improve and correct…

Subjective feedback (Useful for forming a coding style)

Checking code again

Earlier version

python_file.py

```
1  class furniture:
2      def _init_(self, room):
3          self.room = room
4
5  my_chair = char('kitchen')
6
7  def   pricing(furniture, store):
8      furniture.price = 230
9
10 pricing(chair('kitchen'), store1)
```

Newer version

python_file.py

```
1  """
2  Advanced Python Lecture 11 tutorial
3  """
4
5  class Furniture:
6      """ This is a class """
7      def _init_(self, room):
8          self.room = room
9
10 my_chair = Furniture('kitchen')
11
12 def   pricing(furniture, store):
13     furniture.price = 230
14
15 pricing(chair('kitchen'), store1)
```

A few more corrections later…

python_file.py

```
"""
Advanced Python Lecture 11 tutorial
"""

class Furniture:
    """ This is a class """
    def __init__(self, room):
        self.room = room

MY_CHAIR = Furniture('kitchen')

def pricing(furniture):
    """ This is a function """
    furniture.price = 230

pricing(Furniture('kitchen'))
```

```
staff-205-62:Structuring athina$ pylint python_file.py
No config file found, using default configuration
************* Module python_file
R:  5, 0: Too few public methods (0/2) (too-few-public-methods)

--------------------------------------------------------------------
Your code has been rated at 8.57/10 (previous run: 7.14/10, +1.43)
```

Our code has improved:

- Naming of constants
- Docstrings for functions and classes
- No unused variables

How do we ignore recommendations?

Ignoring suggestions

Several suggestions may be subjective
To ignore them we can add a comment in the line where they appear, example:

```
#pylint: disable=too-few-public-methods
```

10/10 !

```
staff-205-62:Structuring athina$ pylint python_file.py
No config file found, using default configuration

--------------------------------------------------------------------
Your code has been rated at 10.00/10 (previous run: 8.57/10, +1.43)
```

python_file.py

```
"""
Advanced Python Lecture 11 tutorial
"""

class Furniture: #pylint: disable=too-few-public-methods
    """ This is a class """
    def __init__(self, room):
        self.room = room

MY_CHAIR = Furniture('kitchen')

def pricing(furniture):
    """ This is a function """
    furniture.price = 230

pricing(Furniture('kitchen'))

```

Summary: Static analysis of code

Analyzing code **without** executing it

Fast to perform; little effort to fix; it can uncover common mistakes

--> Ensure conformity of our code with coding styles (PEP-8)
Improves homogeneity in a coding project, especially useful when multiple contributors work on the same code

Several suggestions may be subjective

The score we receive as output is also subjective

Outline for today

1. Coding principles for writing efficient code
2. Coding styles
3. Source Code Analysis