University of Sheffield

DESIGN YOUR CODE…AND WRITE LESS OF IT!

FAIR^2 training program

R. Thomas and M. Dyer RSE team

Who are we?

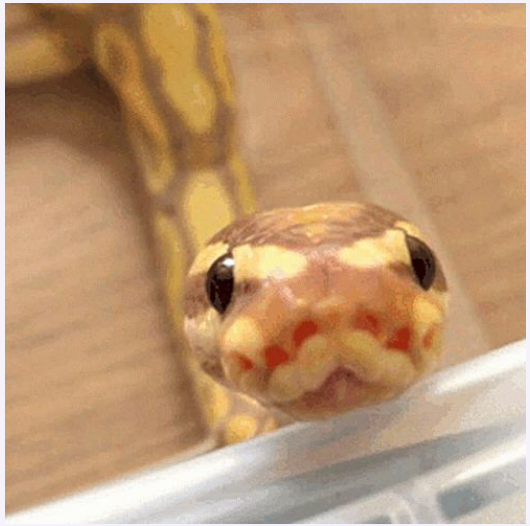
Romain Thomas
romain.thomas@sheffield.ac.uk
Head of RSE

Martin Dyer
martin.dyer@sheffield.ac.uk
Research Software Engineer

Before we start:

For this course we will be using Python.
You should have:
- Python 3.9+
- Pylint 3.0.3+

–No need for extra packages–



What are we going to talk about?

- FAIR principles: reminders [Today]
- Why should you care? [Today]
  - *Learn about readability, reusability and maintainability*
- Code Structure [Today]
  - *Learn to gather functionalities in function/classes and organise your code*
- The Zen of Python [Today]
  - *Learn to write code correctly with the proper syntax*
- Principles of Code Design [Tomorrow]
  - *Learn how to recognise problems in a code and how to tackle them*
- Stop touching your code!!!!!! [Tomorrow]
  - *Learn how to stop touching your code when you run it with different input!*

How are we going to talk about?

Code is read more often than it is written.

- Guido Van Rossum, Creator of Python

The goal of this lecture is to make you think about code.

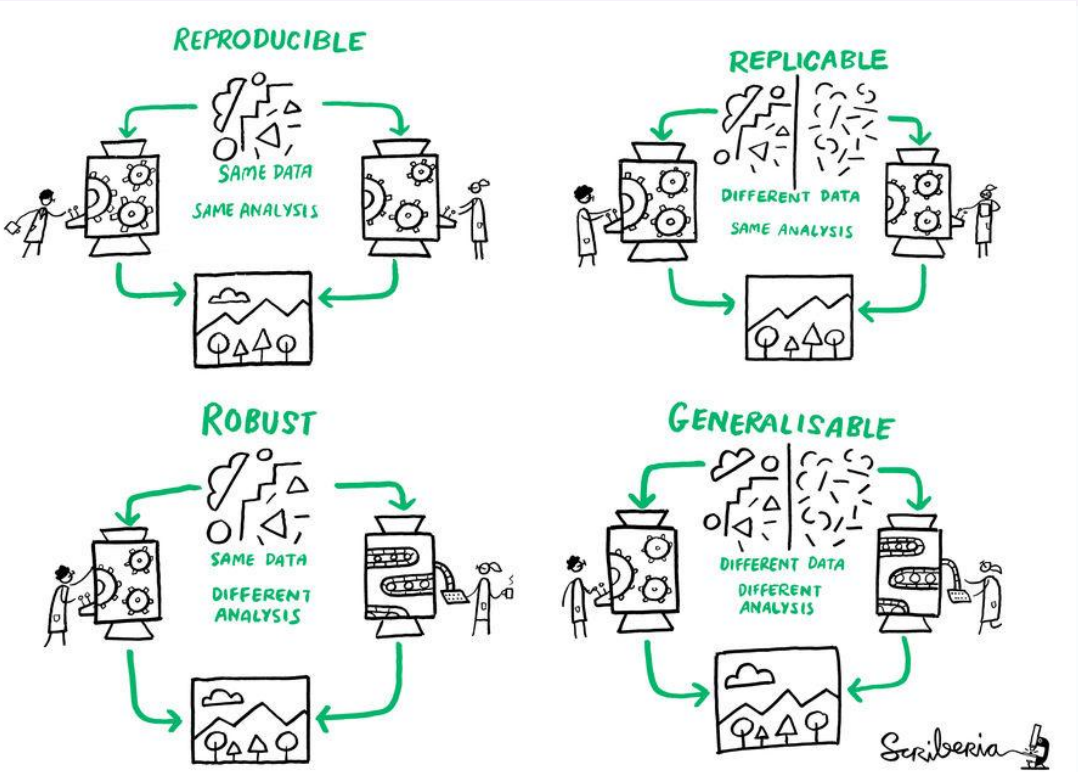
- We will read code together
- We will write code together

Reminder: What is a research software?

“Research Software includes source code files, algorithms, scripts, computational workflows and executables that were created during the research process or for a research purpose.”

Barker et al. Scientific Data 9:622 (2022) “Introducing the FAIR Principles for research software”

Reminders: The FAIR principles



Software is an essential part of research

Sharing this important aspect of the research process with others is *vital* for **reproducible** and **open research**

Image by Scriberia for The Turing Way community, used under a CC-BY license https://zenodo.org/record/3332808

Reminders: The FAIR principles

- **Findable** Software, and its associated metadata, is easy for both humans and machines to find

- **Accessible** Software, and its metadata, is retrievable via standardised protocols

Barker *et al. Scientific Data* 9:622 (2022) “Introducing the FAIR Principles for research software” DOI: 10.1038/s41597-022-01710-x

Reminders: The FAIR principles

- **Interoperable** Software interoperates with other software by exchanging data and/or metadata, and/or through interaction via application programming interfaces (APIs), described through standards
    - → Writing code in a modular way and using standard coding practices allows other people and other system to us it
- **Reusable** Software is both usable (can be executed) and reusable (can be understood, modified, built upon, or incorporated into other software)
    - → Use of docstrings, comments, and documentation makes it easier for others (AND YOU!) to understand, use and modify the code.

Barker *et al. Scientific Data* 9:622 (2022) “Introducing the FAIR Principles for research software” DOI: 10.1038/s41597-022-01710-x

Reminders: The FAIR principles at UoS

‘‘We aspire to open research culture that values a diverse range of contributions and adheres to the FAIR principles to enable the results of our research to be of maximum benefit to society (findable, accessible, interoperable and reusable), whilst also respecting circumstances that limit data sharing (for example, due to issues of privacy, non-consent, contractual agreements, legislation or practicality).’’

University of Sheffield, Statement on Open Research

https://www.sheffield.ac.uk/openresearch/university-statement-open-research

Reminders: The FAIR principles

"Open source software ideally should start working toward satisfying the FAIR4RS Principles ***when it is initially being developed"***

Barker *et al. Scientific Data* 9:622 (2022) "Introducing the FAIR Principles for research software" DOI: 10.1038/s41597-022-01710-x

Why should you care?

> *Knowledge is humankind's most precious treasure. Everything that we accomplished has been done due to the capacity to create a transmisible heritage, which spare each new generation the task of starting from scratch. -B. Sirbey*

Reproducibility and Reliability: In order for research results to be reliable and trusted, they have to be shared.

→ Sharing clean, well-structured, easy to read and use code will increase the reliability of your research

Why should you care?

Efficiency and maintainability: A research project (and code) can span several years and involve multiple contributors. You should not have to take three days each time you try to understand the piece of code from a colleague (or yours)

→ Clean, well structured, easy to read and use code reduces times and effort to understand, troubleshoot, implement new features or adapt it.

Collaboration and community contribution:

→ Clean, well structured, easy to read and use code is welcoming and will attract people!
→ More collaboration, more robust, more ideas, more research…

Why should you care?

Your research might rely heavily on your code.

Writing clean and well structured code

Increase the quality of your research, your reputation and your impact on your research domain

Some definitions: *Readability*

Readability in software refers to how easily a human reader can understand the purpose, control flow, and operation of the code.

Key aspects: *Descriptive namings, consistent formatting, comments, code structure*

Benefits:
-*Maintainability:* Easier to understand and modify, reduce risk of errors
-*Collaboration:* Enhance teamwork and make it easy for others to contribute
-*Efficiency:* Saves a lot of time
-*Quality:* Reduces risk of errors

Some definitions: *Reusability*

Reusability in software refers to the ability to use existing software components across multiple projects without significant modifications. Design to be generic and flexible.

Key aspects: *Modularity, abstraction, generic function, parametrization, no hardcoded values*

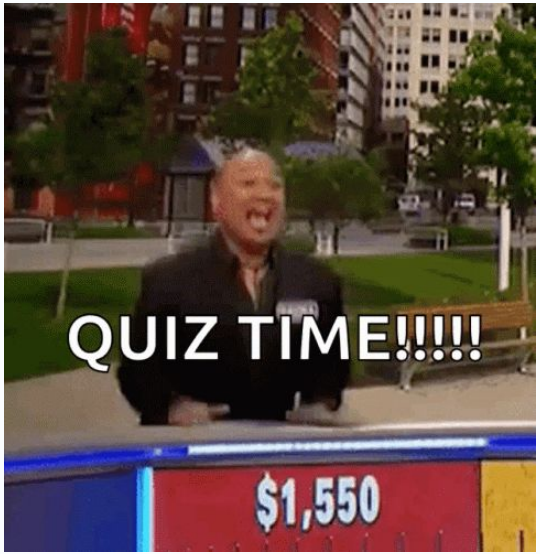
Benefits:
-*Time saving:* Saves dev time, no need to rewrite from scratch
-*Consistency:* Reusing same code across different projects ensures consistency in functionality and behavior
-*Maintainability:* Reusable components can be maintained and updated independently

Some definitions: *Maintainability*

Maintainability in software refers to the ease with which a software can be modified to correct faults, improve performance and adapt to change

Key aspects: *Modularity, readability, testability, reusability*

Benefits:
-Lower maintenance: Easier to refactor and improve over time
-Flexibility: Easier to integrate new features and make updates.
-Productivity: Can be easily understood, spend more time working on new features



We are going to watch two pieces of code. For each of them you will have have to tell me if the code is

Readable, maintainable, and reusable

5 minutes / code

Code Quiz #1

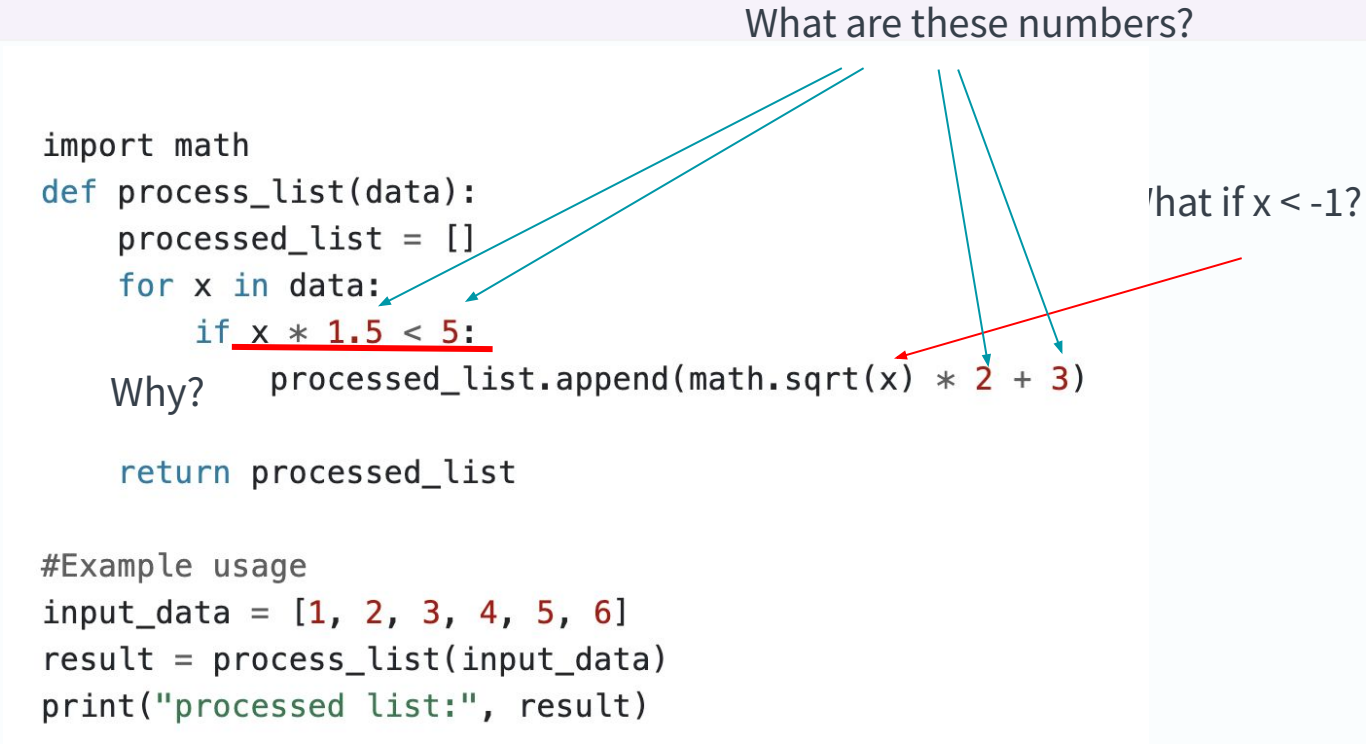
```
import math
def process_list(data):
    processed_list = []
    for x in data:
        if x * 1.5 < 5:
            processed_list.append(math.sqrt(x) * 2 + 3)

    return processed_list

#Example usage
input_data = [1, 2, 3, 4, 5, 6]
result = process_list(input_data)
print("processed list:", result)
```

Code Quiz #1

```
import math
def process_list(data):
    processed_list = []
    for x in data:
        if x * 1.5 < 5:
            processed_list.append(math.sqrt(x) * 2 + 3)

    return processed_list

#Example usage
input_data = [1, 2, 3, 4, 5, 6]
result = process_list(input_data)
print("processed list:", result)
```

- ***Reusable:*** The function can be used with any list of integers to filter and transform the data.
- ***Partially readable:*** Uses a simple structure that is easy to follow. But it is impossible to understand its purpose. There are no comments explaining what the function is doing or why it's doing it.

Code Quiz #1




```python
import math
def process_list(data):
    processed_list = []
    for x in data:
        if x * 1.5 < 5:
            processed_list.append(math.sqrt(x) * 2 + 3)

    return processed_list

#Example usage
input_data = [1, 2, 3, 4, 5, 6]
result = process_list(input_data)
print("processed list:", result)
```

Not maintainable!!!

Code Quiz #2

```
def calculate_statistics():
    data = [23, 45, 12, 67, 34, 89, 23, 45, 23, 34]
    total_sum = sum(data)
    count = len(data)
    average = total_sum / count

    data_sorted = sorted(data)
    if count % 2 == 0:
        median = (data_sorted[count // 2 - 1] + data_sorted[count // 2]) / 2
    else:
        median = data_sorted[count // 2]

    occurrences = {}
    for num in data:
        if num in occurrences:
            occurrences[num] += 1
        else:
            occurrences[num] = 1
    mode = max(occurrences, key=occurrences.get)

    print("Sum:", total_sum)
    print("Average:", average)
    print("Median:", median)
    print("Mode:", mode)

# Calculate statistics for the specific data set
calculate_statistics()
```

Code Quiz #2

- ***Maintainable***: The code is well-structured, with clear variable names and straightforward logic. It's easy to understand and modify if needed.

- **Readable**: The code uses descriptive variable names and simple constructs, making it easy to follow.

```
def calculate_statistics():
    data = [23, 45, 12, 67, 34, 89, 23, 45, 23, 34]
    total_sum = sum(data)
    count = len(data)
    average = total_sum / count

    data_sorted = sorted(data)
    if count % 2 == 0:
        median = (data_sorted[count // 2 - 1] + data_sorted[count // 2]) / 2
    else:
        median = data_sorted[count // 2]

    occurrences = {}
    for num in data:
        if num in occurrences:
            occurrences[num] += 1
        else:
            occurrences[num] = 1
    mode = max(occurrences, key=occurrences.get)

    print("Sum:", total_sum)
    print("Average:", average)
    print("Median:", median)
    print("Mode:", mode)

# Calculate statistics for the specific data set
calculate_statistics()
```

Code Quiz #2

It is ***NOT reusable*** because the function is **hardcoded to work with a specific dataset defined within the function**.

It cannot be easily reused with different datasets without modifying the function itself.

```
def calculate_statistics():
    data = [23, 45, 12, 67, 34, 89, 23, 45, 23, 34]
    total_sum = sum(data)
    count = len(data)
    average = total_sum / count

    data_sorted = sorted(data)
    if count % 2 == 0:
        median = (data_sorted[count // 2 - 1] + data_sorted[count // 2]) / 2
    else:
        median = data_sorted[count // 2]

    occurrences = {}
    for num in data:
        if num in occurrences:
            occurrences[num] += 1
        else:
            occurrences[num] = 1
    mode = max(occurrences, key=occurrences.get)

    print("Sum:", total_sum)
    print("Average:", average)
    print("Median:", median)
    print("Mode:", mode)

# Calculate statistics for the specific data set
calculate_statistics()
```

Code structure

When you're writing code, making it consistent and well-structured is just as essential as ensuring it produces the correct result.

You should think about how you structure your code as you write it, as this will make it easier to read and maintain in the future both by yourself and others.

It's much easier to get other people to use and contribute to your code if it's easy to read!

Objects and Functions

Python is an **object-oriented** programming language.

You should try to think about your code as being made up of **objects** (numbers, text, lists etc), and **functions** that act upon those objects (and usually return a new object as the output).

For example, we create a function called `add()`, which takes two numbers and returns the sum. We also use the built-in `print()` function to print the result.

```
def add(a, b):
    return a + b

result = add(3, 5)
print(result)
```

Scope

A variable defined within a function can only be accessed within that function. This is the **scope** of the variable.

Any variable outside of a function is in the **global scope**, and can be accessed from anywhere (including within functions).

Structure Challenge #1: What will this code output?

```
x = 5
y = 10

def my_function(z):
    x = 20
    return x + y + z
result = my_function(3)

print('x:', x)
print('y:', y)
print('Result:', result)
print('z:', z)
```

Classes

Every Python object has a **class**, like `int` for integers or `str` for text.

You can also create your own classes, which can have **properties** and **methods**.

Here we make a simple `Rectangle` class, which has properties for width and height and a method to find its area (note the use of the `self` keyword).

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def get_area(self):
        return self.width * self.height
```

Classes

Each time you call a class you create a new **instance**, which can have different properties.
You can then access the properties and methods using the `.` operator:

```
my_rectangle = Rectangle(10, 20)
print('This rectangle has a width of', my_rectangle.width, 'and a height of', my_rectangle.height)
print('Its area is', my_rectangle.get_area())
```

OUTPUT

```
This rectangle has a width of 10 and a height of 20
Its area is 200
```

Classes

Structure Challenge #2

The code below defines various items in stock at a grocery store. Each item has a name, price, and the number in stock.

```
fruits = ['apple', 'banana', 'orange']
fruit_prices = [1.00, 0.50, 0.75]
fruit_count = [10, 20, 15]
```

Create a `Stock` class with `name`, `price` and `count` properties, plus a method `display()` to print the name and price, and a method to `get_total_value()` which returns `price` times `count`.

Classes

Structure Challenge #3

Now create a `Shop` class, which takes as an input a list of `Stock` objects.

It should have two methods:

- `display_stock()`, which calls the `display()` method on each item in the stock list
- `get_total_stock_value()`, which returns the total value of all items in the `stock` list

Scripts and Modules

Python **scripts** are files that are executed by the Python interpreter on the command line, e.g. `$ python my_script.py`.

Once your code is too large for a single script, you can put functions and classes in separate files (**modules**) and then import them into the script.

module:

```
"""calculator.py"""

def add(a, b):
    return a + b
```

script:

```
"""script.py"""
import calculator

result = calculator.add(5, 3)
print(result)
```

command line:

```
$ python script.py
8
```

Executing code in modules

Any code in a module is executed when the module is imported.

If you still want some lines to be called when the file is run as a script, you can use the special check **if __name__ == '__main__':**

```
"""calculator.py"""
def add(a, b):
    return a + b

if __name__ == '__main__':
    result = add(5, 3)
    print('Test result:', result)
```

Packages

A collection of scripts and modules is called a **package**.

Packages are organised into directories which contain a file called `__init__.py`.

This file doesn't have to contain anything, but it works to mark a directory as containing modules to import.

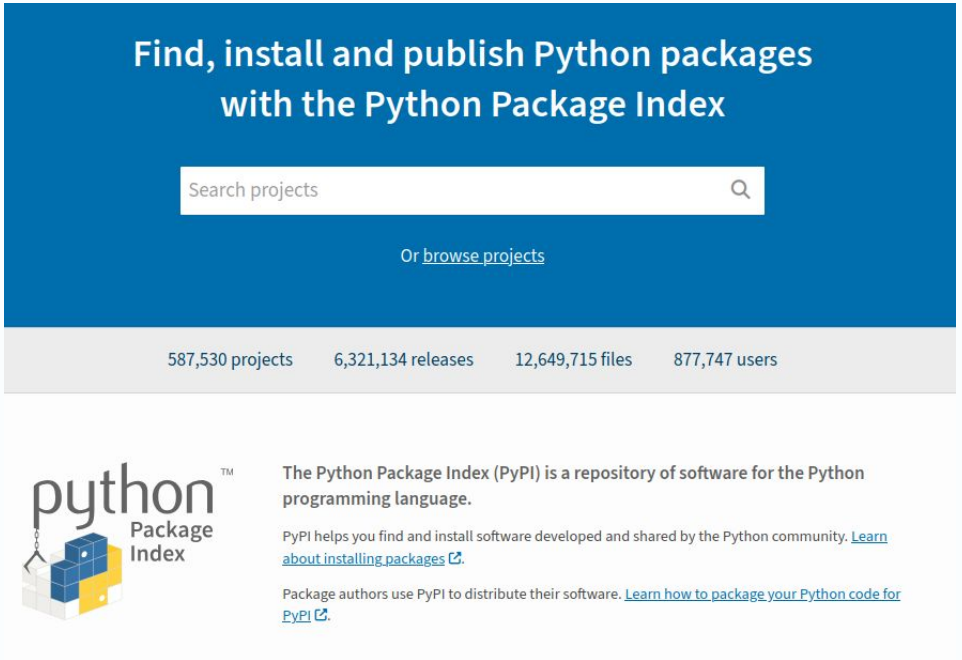
```
my_package/
    __init__.py
    calculator.py
    geometry.py
```

```
from my_package import calculator
calculator.add(5, 3)
```

Packages

Python has a huge number of packages on the Python Package Index (https://pypi.org), which can be easily downloaded and installed using `pip` or `conda`.

We will cover how to create a formal Python package and share it with others in a future FAIR² session.



Packages

Structure Challenge #4

A single Python file contains the following functions and classes:

- `load_data()`: a function that reads data from a file
- `clean_data()`: a function that removes any missing values from the data
- `plot_data()`: a function that plots the data
- `Data`: a class that holds the data, returned by `load_data()`
- `Model`: a class that represents a machine learning model created from the data
- `test_data()`: a function that tests the data class is working correctly
- `test_model()`: a function that tests the model class is working correctly
- `run_experiment()`: a function that runs the entire experiment, taking a data file as an input

How would you organise this into a package with multiple modules?

Code structure final exercise

Download this Python script:
https://fair2-for-research-software.github.io/FAIR_Code_design/files/student_scores.py

The script reads in data about a class of students, and prints out various results like their total scores, the average score per exercise and which student is leading the class.

Hopefully you should be able to see how to rewrite the code to use a data class and replace the repetitive code with common functions and methods.

https://elpythonista.com/zen-of-python

PEPs: Python Enhancement Proposals

→ Design documents that provide information to the Python community about new features, processes or environments.

Some of them are dedicated to design and style considerations:

- PEP8 : Rules to write out Python code
- PEP257: Convention to write docstrings

PEP 8 – Style Guide for Python Code

Author: Guido van Rossum <guido at python.org>, Barry Warsaw <barry at python.org>, Alyssa Coghlan <ncoghlan at gmail.com>
Status: Active
Type: Process
Created: 05-Jul-2001
Post-History: 05-Jul-2001, 01-Aug-2013

▸ **Table of Contents**

Introduction

This document gives coding conventions for the Python code comprising the standard library in the main Python distribution. Please see the companion informational PEP describing style guidelines for the C code in the C implementation of Python.

This document and PEP 257 (Docstring Conventions) were adapted from Guido's original Python Style Guide essay, with some additions from Barry's style guide [2].

This style guide evolves over time as additional conventions are identified and past conventions are rendered obsolete by changes in the language itself.

Many projects have their own coding style guidelines. In the event of any conflicts, such project-specific guides take precedence for that project.

PEP20 : The Zen of Python

Python easter egg. It proposes 20 aphorisms that ***guide the development of Python software***

Is is accessible from the python interpreter.

```
>>> 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!
>>>
```

The Zen of Python: Readability count

> A code is read much more often then it is written
>
> Guido van Rossum, BDFL of Python

AUTHOR: Karlisson M. Bezerra, hacktoon.com

You might spend hours/days writing a piece of code. Then you will never write it again. But you might have to read it again and again and again. Understanding a code after a few weeks/months is not easy.

→ *Following standard guidelines will greatly help you and save a lot of time!*

The Zen of Python: Readability count

So how do we make a code more readable?

The Zen of Python: Explicit is better than implicit

AUTHOR: Karlisson M. Bezerra, hacktoon.com

1st step:

Write code that is meaningful!
Give sense to all quantity that you write in your code

```
# This is bad:

x = 5
y = 10
z = 2*x + 2*y

# This is much better:

width = 5
height = 10
diameter = 2*height + 2*width
```

The Zen of Python: Explicit is better than implicit

2nd step:

Everything you write has a way to be written

Variable, function and methods use the **snake_case** convention

→ *Lowercase, words separated by underscores.*

```
# This is bad
def ComputeDiameter(width, height):
    return 2*width + 2*height

# This is good
def compute_diameter(width, height):
    return 2*width + 2*height
```

The Zen of Python: Explicit is better than implicit

2nd step:

Everything you write has a way to be written

Constant names follow the **UPPER_SNAKE_CASE** convention

→ Each word is written in capital letters, words are separated by underscores

```
# This is bad:
speedoflight = 3e8
plankconstant = 6.62e-34

# This is good
SPEED_OF_LIGHT = 3e8
PLANK_CONSTANT = 6.62e-34
```

The Zen of Python: Explicit is better than implicit

2nd step:

Everything you write has a way to be written

```
# This is bad
class example_class:

# This is good
class ExampleClass:
```

Class names follow the **PascalCase (or CamelCase)** convention

→ Each words start with Capital, words are **NOT** separated by underscores.

The Zen of Python: Beautiful is better than ugly

3rd step:

Always try to write code that is easy to understand (=Beautiful!)

AUTHOR: Karlisson M. Bezerra, hacktoon.com

You should go for simple solution. If you try to cram as much functionalities as possible in a single line it will takes ages to understand.. **Break down the code in simple components.**

The Zen of Python: Beautiful is better than ugly

3rd step:

Always try to write code that is easy to understand (=Beautiful!)

You should go for simple solution. If you try to cram as much functionalities as possible in a single line it will takes ages to understand.. **Break down the code in simple components.**

Zen Challenge #1

What is this code doing?

PYTHON < >

```python
result = [x * 2 if x % 2 == 0 else x * 3 for x in range(10) if x % 2 != 1 and x != 4]
```

The Zen of Python: Beautiful is better than ugly

3rd step:

Always try to write code that is easy to understand (=Beautiful!)

You should go for simple solution. If you try to cram as much functionalities as possible in a single line it will takes ages to understand.. **Break down the code in simple components.**

Zen Challenge #2: Rewrite it to make it more understandable

What is this code doing?

PYTHON < >

```python
result = [x * 2 if x % 2 == 0 else x * 3 for x in range(10) if x % 2 != 1 and x != 4]
```

The Zen of Python: Beautiful is better than ugly

3rd step:

Always try to write code that is easy to understand (=Beautiful!)

Which does not mean you should over-complexify things…

Zen Challenge #3:
Rewrite the following code in two lines

```
def is_empty(lst):
    if len(lst) == 0:
        return True
    else:
        return False

lst = []
print(is_empty(lst))
```

The Zen of Python: Sparse is better than dense

Zen Challenge #4:

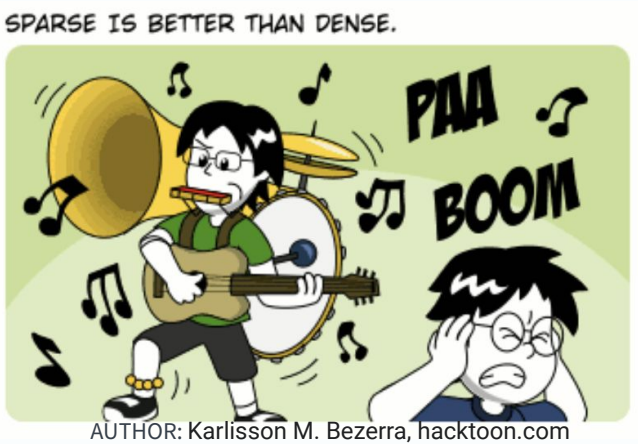
Is it actually working?

```
def   example_function(param1,param2):print(param1+param2*2)
def   another_function(x,y):return x+y
class MyClass:
    def __init__(self,param): self.param=param
    def  method(self):
        if self.param >10: print("Value is greater than 10")
        else: print("Value 10")
my_list=[1,2,3,4,5]
dictionary={'key1':'value1','key2':'value2'}
result=another_function(5,10)
print(result)
```

The Zen of Python: Sparse is better than dense

Fourth step:

- Avoid cluttered code by making it sparse and spaced out.
- Use whitepaces, correct indentations and separation



AUTHOR: Karlisson M. Bezerra, hacktoon.com

Rules:

- Indentation: 4 spaces (don't use tabs)

```
def example_function():
    if True:
        print("Indented correctly")
```

The Zen of Python: Sparse is better than dense

So what are the rules?

Fourth step:

Make your code easy to read, include spaces!!!

- Avoid cluttered code by making it sparse and spaced out.
- Use whitepaces, correct indentations and separation

Rules:

- Indentation: 4 spaces (don't use tabs)
- Whitespaces around operators

```
#This is bad
a=2
b=3
c=4
result=a+b+c


#This is good
a = 2
b = 3
c = 4
result = a + b * c
```

The Zen of Python: Sparse is better than dense

So what are the rules?

Fourth step:

Make your code easy to read, include spaces!!!

- Avoid cluttered code by making it sparse and spaced out.
- Use whitepaces, correct indentations and separation

Rules:

- Indentation: 4 spaces (don't use tabs)
- Whitespaces around operators
- Comma and colon spacing: single space after common, space after the colon

```
#This is bad
dictionary={'key1':'value1','key2':'value2'}

#This is good
dictionary = {'key1': 'value1', 'key2': 'value2'}
```

The Zen of Python: Sparse is better than dense

So what are the rules?

Fourth step:

Make your code easy to read, include spaces!!!

- Avoid cluttered code by making it sparse and spaced out.
- Use whitepaces, correct indentations and separation

Rules:

- Indentation: 4 spaces (don't use tabs)
- Whitespaces around operators
- Comma and colon spacing: single space after common, space after the colon
- Blank lines: 2 blank lines before a top level function or class definition, single blank line between method definitions

The Zen of Python: Sparse is better than dense

Zen Challenge #5: Create a new python file and correct the syntax based on the previous rules

```
def   example_function(param1,param2):print(param1+param2*2)
def   another_function(x,y):return x+y
class MyClass:
    def __init__(self,param): self.param=param
    def  method(self):
        if self.param >10: print("Value is greater than 10")
        else: print("Value 10")
my_list=[1,2,3,4,5]
dictionary={'key1':'value1','key2':'value2'}
result=another_function(5,10)
print(result)
```

The Zen of Python: If the implementation is hard to explain it is bad

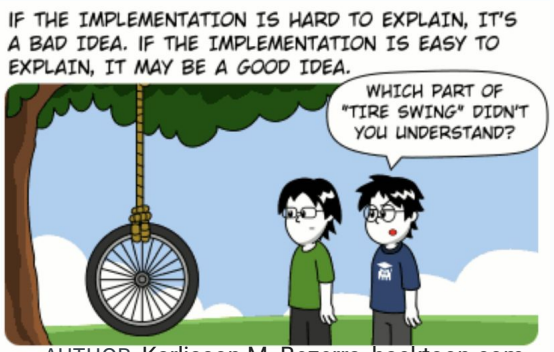
Fifth step:

Writing simpler and more logical code is a good starting point. **But adding comments is very important!** Few rules apply:

- Comments should be (real) sentences
- Block comments apply to the piece of coming after them
- Comments should not state the obvious (don't make distraction)

Keep your comments up to date!

→ 'Comments contradicting the code are worse than no comments'



AUTHOR: Karlisson M. Bezerra, hacktoon.com

The Zen of Python: Don’t want to remember all this?

Pylint is a tool that **analyzes Python code to find programming errors, enforce a coding standard, and look for improvements.** It provides a score based on the number of issues detected, helping you writing clean and readable code.

The Zen of Python: Don’t want to remember all this?

It will identify Errors in Syntax, undefined variables, useless imports, highligh duplicated code, etc… It will check the code against PEP8 and verify all the coding standards that we just discussed.

To run Pylint on your code it is as simple as

```
pylint your_python_file.py
```

The Zen of Python: Don’t want to remember all this?

It will identify Errors in Syntax, undefined variables, useless imports, highligh duplicated code, etc… It will check the code against PEP8 and verify all the coding standards that we just discussed.

To run Pylint on your code it is as simple as

```
pylint your_python_file.py
```

Zen Challenge #6: Take the last code, apply pylint and clean it up!

The Zen of Python: Don’t want to remember all this?

To finish:

- It is possible to tune Pylint with a configuration file
- Many Integrated Development Environment support Pylint integration.→ You can see linting results directly as you write the code.
- Github actions can be used to check the code automatically.

Coding Principles

What are coding principles?

Coding principles are ***guidelines and best practices*** that anybody writing code should follow to write clean, maintainable, and efficient code.

These principles ***enhance the quality of code and ensure it is readable, reusable, and less prone to errors.***

What are coding principles?

Coding principles are ***guidelines and best practices*** that anybody writing code should follow to write clean, maintainable, and efficient code.

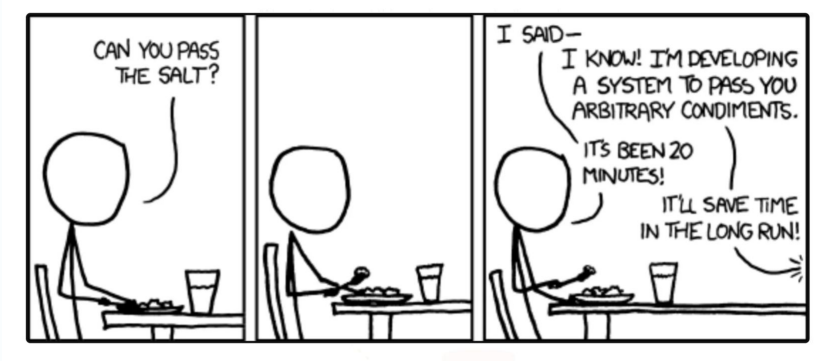
These principles ***enhance the quality of code and ensure it is readable, reusable, and less prone to errors.***

They are all easy to understand and to remember. And they are quite *obvious!*

Today we will have a look at a few of them:

- YAGNI
- KISS and Curly's Law
- DRY and the rule of three
- POLA

YAGNI: You aren’t gonna need it

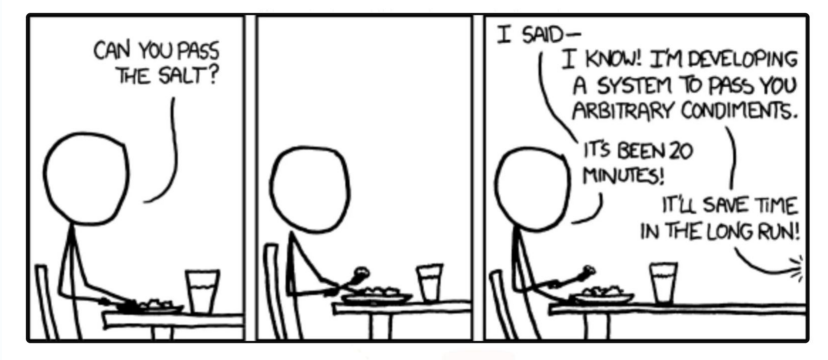


YAGNI encourages you to build ONLY what you need RIGHT NOW.

→ Do not add features for hypothetical future needs.

→ Coming from Agile programming, it aims to reduce wasting time and resources on unnecessary code.

YAGNI: You aren’t gonna need it



Why?

Simplicity: Unnecessary code introduce complexity and is harder to read (Why is this implemented???)

Saving time: Unnecessary code takes time to write!

Flexibility: If you write JUST what is needed it is easier to implement changes

YAGNI: You aren’t gonna need it

An example:

The instruction here are to create a simple function that implements a percentage discount price.

→ Look at the implementation →

What can you say about it with respect to the YAGNI principle?

```python
def calculate_discount(price, discount_type="percentage", value=10.0):
    '''
    This function applies a discount to a price

    Parameters
    ----------
    price   : float
              Original price
    discount_type: str
                   type of discount [percentage of fixed]
    value:  float
            discount to be applied

    Return
    ------
    discounted_price: float
                      final price after applying discount

    Raises
    ------
    ValueError
            if the discount type is not 'percentage' or 'fixed'

    '''
    if discount_type == "percentage":
        return price - (price * (value / 100))
    elif discount_type == "fixed":
        return price - value
    else:
        raise ValueError("Invalid discount type")
```

YAGNI: You aren’t gonna need it

An example:

The instructions here are to create a simple function that implements a percentage discount price.

Here is a better version following the YAGNI principle:

```
def calculate_discount(price, discount_percentage):
    '''
    Function that applies a discount. The discount is given as a percentage of the original price.

    Parameter
    ---------
    price:  float
            original price

    Return
    ------
    final_price: float
                 final price after applying discount
    '''
    final_price = price - (price * (discount_percentage / 100))
    return final_price
```

YAGNI: You aren’t gonna need it

Principle challenge #1

Context:

You’re working on a feature to calculate the final price of items in a shopping cart. Right now, the only two requirements are

- (1) to apply a fixed 10% discount to the total cart price
- (2) return the final price with a $ sign in front of the total price (e.g. $42.2).

```python
def calculate_final_price(prices, currency="USD", discount_type="percentage", discount_value=0.1, include_shipping=False, shipping_cost=5.0):

    # Calculate the initial total price
    total = sum(prices)

    # Apply discount based on type
    if discount_type == "percentage":
        total -= total * discount_value
    elif discount_type == "fixed":
        total -= discount_value

    # Include shipping if specified
    if include_shipping:
        total += shipping_cost

    # Format total with currency symbol
    if currency == "USD":
        return f"${total:.2f}"
    elif currency == "EUR":
        return f"€{total:.2f}"
    else:
        raise ValueError("Unsupported currency")
```

KISS and Curly’s law

KISS stands for ‘Keep it simple, stupid’. It exists to remind you that *writing simple code should be a primary goal* in design.

KISS and Curly’s law

Why?

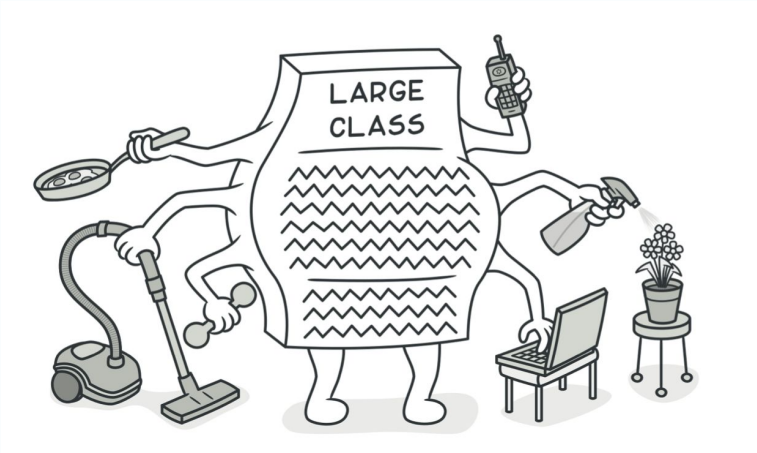
Reusability: Simple single-task function are easier to reuse.

Bug fix: When you code is composed of simple function, potential issues are easier to localise.

Testing: Simple single-task function are easier to test.

Modularity: Code becomes more modular and organized.

KISS and Curly’s law



Curly’s law

This ‘law’ says that each function should ‘do one thing’ and ‘do it well’

→ If a function has multiple tasks, consider breaking it down

KISS and Curly's law

An example:

Let's consider a function that compute the area of circles, rectangles and triangles →

What would you do to make this code more easy?

```
def calculate_area(shape, dimensions):
    '''
    This function compute the area of a given geometrical shape


    Parameters
    ----------
    shape   : str
              shape to consider. Can be rectangle, circle or triangle

    dimensions: list
                of dimension to consider. For rectangle and triangle you need to give a list
                of 2 numbers. For circle, you need to pass a list of one quantity (radius).

    Return
    ------
    area      : float
                area of the shape

    Raises
    ------
    ValueError
            if the shape is not recognised
    '''
    if shape == "rectangle":
        area = dimensions[0] * dimensions[1]
    elif shape == "circle":
        area = 3.14159 * (dimensions[0] ** 2)
    elif shape == "triangle":
        area = 0.5 * dimensions[0] * dimensions[1]
    else:
        raise ValueError("Unsupported shape!")

    return area

area = calculate_area("rectangle", [10, 20])
```

KISS and Curly's law

An example:

In that version, functions are specific and easy to understand and there is no unnecessary complexity in shape management. It is easier to maintain and extend.

```python
def rectangle_area(length, width):
    return length * width

def circle_area(radius):
    return 3.14159 * radius ** 2

def triangle_area(base, height):
    return 0.5 * base * height

# Simple and clear usage
area = rectangle_area(10, 20)
```

KISS and Curly’s law

Principle challenge #2

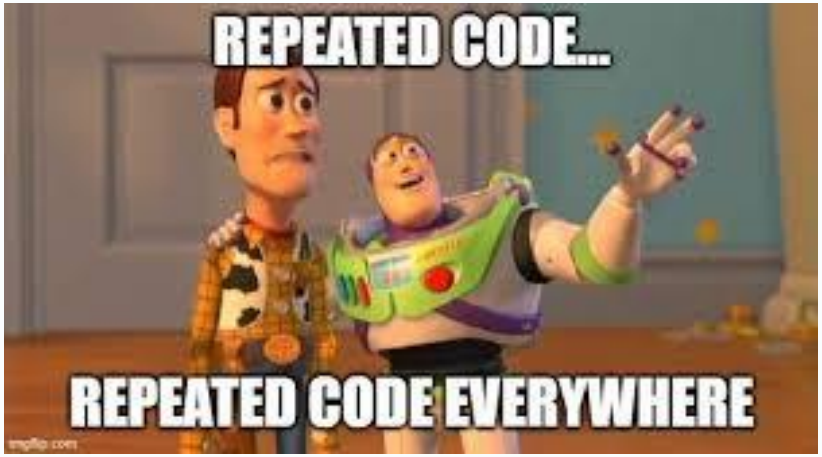
Let's consider a function that processes data by removing values, calculating the average and returning a formatted result :

PYTHON

```python
def process_data(data):
    cleaned_data = [x for x in data if x is not None]  # Remove missing values
    average = sum(cleaned_data) / len(cleaned_data)    # Calculate average
    return f"Average: {average:.2f}"
```

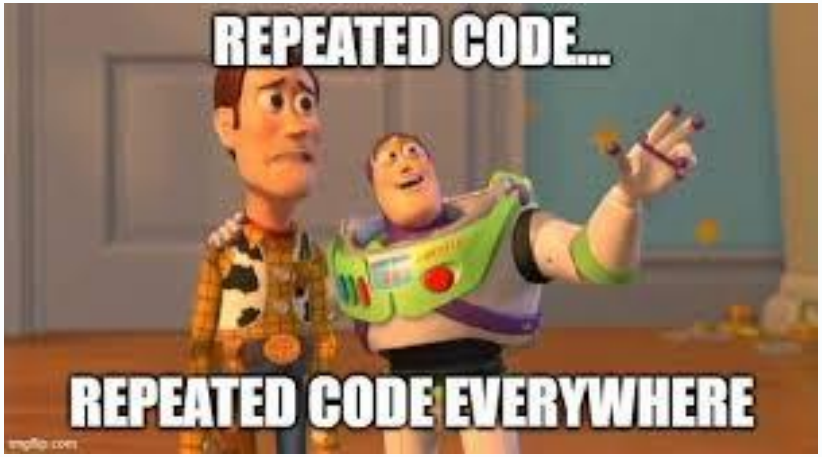
Using KISS and Curly's law, rewrite this code.

DRY and Rule of Three



DRY stands for ‘Don’t repeat yourself’. It encourages you to minimize duplication by refactoring similar code patterns.

DRY and Rule of Three



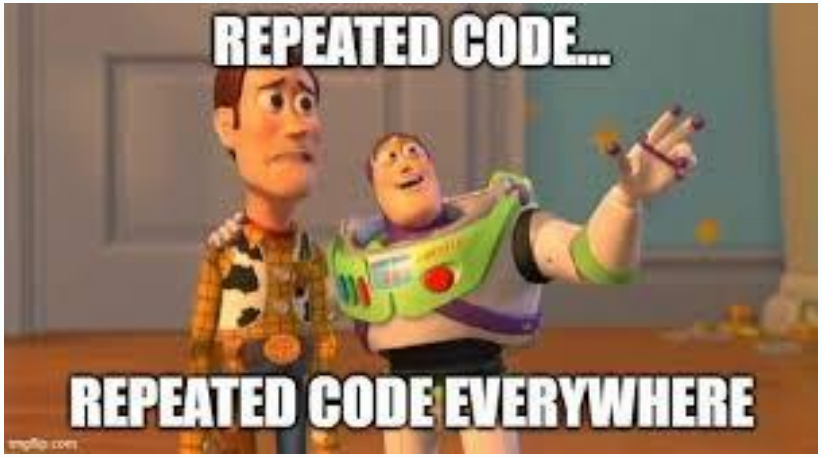
Why?

Improves Readability: Code is clearer when it's not cluttered with repeated part.

Reduces Bugs: If you need to make changes, you only do it in one place, reducing the chance of errors.

Saves Time: Updating and testing code is faster when code is organized with minimal duplication.

DRY and Rule of Three



BUT don’t go too fast!!

Prematurely refactoring a code might lead to unnecessary complexity.

→ Dry is associated with the *Rule of Three*.

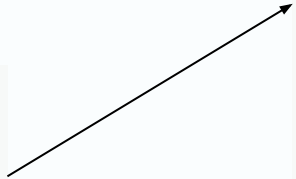
→ You should wait until a piece of code is repeated three times before refactoring it. It ensures that you only refactor when a pattern is stable and repeated enough time

DRY and Rule of Three

In practice:

Using function to refactor code that is the same in multiple places

```
price1 = 100 * 1.2
price2 = 150 * 1.2
price3 = 200 * 1.2
print(price1, price2, price3)
```



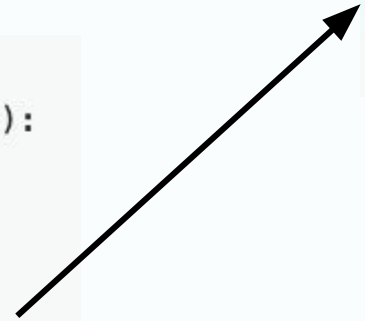
```
# With DRY Principle
def calculate_price(base_price):
    return base_price * 1.2

price1 = calculate_price(100)
price2 = calculate_price(150)
price3 = calculate_price(200)
print(price1, price2, price3)
```

DRY and Rule of Three

In practice:

Using loops instead of manual repetition:

```python
# With DRY Principle
def calculate_price(base_price):
    return base_price * 1.2

price1 = calculate_price(100)
price2 = calculate_price(150)
price3 = calculate_price(200)
print(price1, price2, price3)
```



```python
prices = [100, 150, 200]
for price in prices:
    print(calculate_price(price))
```

DRY and Rule of Three

In practice:

Using constant for common values:

```
total = (100 * 0.1) + (200 * 0.1) + (300 * 0.1)
```



```
TAX_RATE = 0.1
total = (100 * TAX_RATE) + (200 * TAX_RATE) + (300 * TAX_RATE)
```

DRY and Rule of Three

Principle challenge #3

Write a code, without repetition, that produces the following output:

```
Hello, Alice!
Hello, Bob!
Hello, Charlie!
```

POLA: Principle of least astonishment

https://medium.com/@mesw1/the-principle-of-least-astonishment-making-intuitive-software-46d9ae1b8abe

POLA states that the code you are writing should work in a way that does not surprise the users and maintainers.

→ It should align with common expectations.

POLA: Principle of least astonishment

https://medium.com/@mesw1/the-principle-of-least-astonishment-making-intuitive-software-46d9ae1b8abe

Why?

Usability: When code works as expected, users and maintainers are less likely to misuse or misunderstand it.

Maintainability: Familiar and predictable patterns make the code easier to maintain and upgrade.

Collaboration: Using consistent and intuitive code make it easier for multiple people to work with and develop.

POLA: Principle of least astonishment

Common violations:

- Unexpected Return Types: Functions that return types users wouldn't expect, such as a function sometimes returning an integer and other times returning None.

```
def calculate_total(items):
    if not items:
        return None  # If no items, return None
    return sum(items)
```

POLA: Principle of least astonishment

Common violations:

- Multiple Functionalities: Using functions for multiple unrelated tasks often leads to unexpected behaviors.

```python
def process_data(data, save=False):
    cleaned_data = [d.strip() for d in data]
    if save:
        with open('data.txt', 'w') as f:
            f.write('\n'.join(cleaned_data))
    return cleaned_data
```

POLA: Principle of least astonishment

Refactor `calculate_area` to make it more predictable and intuitive.

```
from math import pi

def calculate_area(shape, a, b=0):
    if shape == "rectangle":
        return a * b  # Expects both `a` and `b`
    elif shape == "circle":
        return pi * (a ** 2)  # Ignores `b`
    elif shape == "triangle":
        return 0.5 * a * b  # Expects `a` as base and `b` as height
    else:
        return "Unknown shape"

# Example usage:
print(calculate_area("rectangle", 5))
print(calculate_area("circle", 3, 4))
print(calculate_area("triangle", 6, 3))
print(calculate_area("hexagon", 5, 5))
```

Principle challenge #4

Don’t touch your code anymore!

Don’t touch your code anymore!

When coding for a research project you might find yourself trying to rerun your code with different settings.

Two solutions:

- Change these parameters in the code itself.

- Get the setting of these parameters out of the code.

Don’t touch your code anymore!

Two solutions:

- Change these parameters in the code itself.

- Get the setting of these parameters out of the code.

- Configuration files

- Command line interfaces [CLI]

Don’t touch your code anymore!

Configuration file [CF] is a generally a text file with a particular structure that allows you to set up the initial configuration of your software.

```
#This is an example of configuration file
[Project_info]
Name = to_be_changed
Directory = to_be_changed


[Conf]
main_window_width = 1150
main_window_height = 700
zoom_window_width = 900
zoom_window_height = 400
cluster_window_width = 900
cluster_window_height = 400
compare_window_width = 900
compare_window_height = 400



[Options]
Image_width = 200
Extensions = .tif;.png
Downgrade_factor = 10
```

Don’t touch your code anymore!

Easier reproducibility: saving the CF you can reproduce previous results later.

Minimize code modification: Parameters are externalised, code is cleaner and more maintainable.

Collaboration: it is easy to share, and people can adjust the use of your code for their own need

Documentation: It serves as documentation for your run

Git: you can version your CF along your code.

```
#This is an example of configuration file
[Project_info]
Name = to_be_changed
Directory = to_be_changed


[Conf]
main_window_width = 1150
main_window_height = 700
zoom_window_width = 900
zoom_window_height = 400
cluster_window_width = 900
cluster_window_height = 400
compare_window_width = 900
compare_window_height = 400



[Options]
Image_width = 200
Extensions = .tif;.png
Downgrade_factor = 10
```

Don’t touch your code anymore!

Multiple types of configuration files exists in Python:

INI file: Easy to read and parse. Can be done without extra library installation. Everything is a string.

TOML file:

JSON file:

YAML file

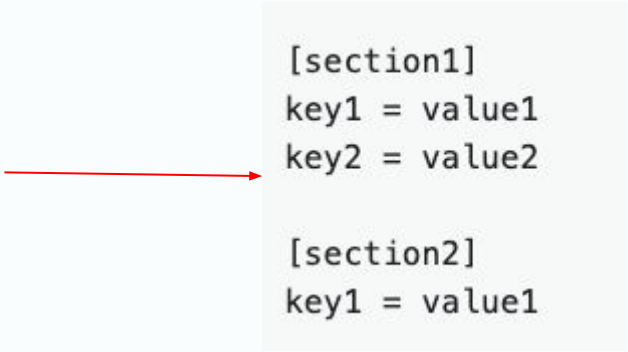
```
[section1]
key1 = value1
key2 = value2

#Comments

[section2]
key1 = value1


[Section3]
key = value3
    multiline
```

Don’t touch your code anymore!

Multiple types of configuration files exists in Python:

INI file: Easy to read and parse. Can be done without extra library installation. Everything is a string.

TOML file: A bit more recent, they allow structure and understand data format. Available from Python 3.11.



```
[section1]
key1 = value1
key2 = value2

[section2]
key1 = value1
```

JSON file:

YAML file

Don’t touch your code anymore!

Multiple types of configuration files exists in Python:

INI file: Easy to read and parse. Can be done without extra library installation. Everything is a string.

TOML file: A bit more recent, they allow structure and understand data format. Available from Python 3.11.

JSON file: Popular in Web apps, they are [a little] more complex to write. Do not allow comments.

YAML file

Don’t touch your code anymore!

Multiple types of configuration files exists in Python:

INI file: Easy to read and parse. Can be done without extra library installation. Everything is a string.

TOML file: A bit more recent, they allow structure and understand data format. Available from Python 3.11.

JSON file: Popular in Web apps, they are [a little] more complex to write. Do not allow comments.

YAML file: Easy to write, allow structure and very popular (Github actions use them). Not available in the Standard library

```
section1:
  key1: value1
  key2: value2

section2:
  key1: value1

# Comments
```

Don’t touch your code anymore!

Conf and CLI challenge #1

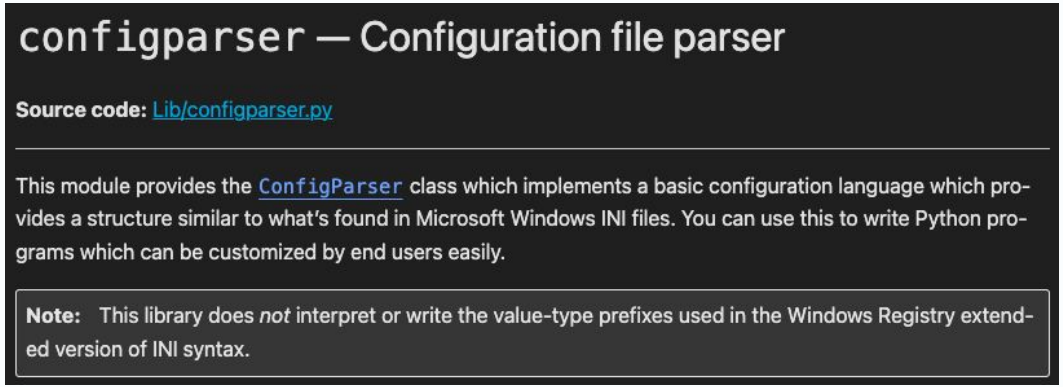
Using the text editor of your choice, create an INI file with three sections: simulation, environment and initial conditions. In the first section, two parameters are given: `time_step` set at 0.01s and `total_time` set at 100.0s. The environment section also has two parameters with `gravity` at 9.81 and `air_resistance` at 0.02. Finally the initial conditions are: `velocity` at 10.0 km/s, `angle` at 45 degrees and `height` at 1m.

Don’t touch your code anymore!

Using INI configuration files in Python

Python gives you the *configparser* library. Allows you to read and write configuration file in a very easy way

→ Part of the standard library.



`configparser` — Configuration file parser

Source code: Lib/configparser.py

This module provides the `ConfigParser` class which implements a basic configuration language which provides a structure similar to what’s found in Microsoft Windows INI files. You can use this to write Python programs which can be customized by end users easily.

Note: This library does *not* interpret or write the value-type prefixes used in the Windows Registry extended version of INI syntax.

Don’t touch your code anymore!

Reading an INI configuration file is very easy and can be done in three lines:

```
##Import the library
import configparser

##Create the parser object
parser = configparser.ConfigParser()

##Read the configuration file
parser.read('config.ini')
```

Don’t touch your code anymore!

From here you can access all the data inside the file:

```
>>> print(parser.sections())
['simulation', 'environment', 'initial_conditions']
```

Get all the sections

Get all keyword names in a section

```
>>> options = parser.options('simulation')
['time_step', 'total_time']
```

```
>>> items_in_simulation = parser.items('simulation')
>>> print(items_in_simulation)
[('time_step', '0.01'), ('total_time', '100.0')]
```

Get all pairs in a section

Don’t touch your code anymore!

From here you can access all the data inside the file:

Turn a section into a dictionary

```
>>> dict(parser['simulation'])
{'time_step': '0.01', 'total_time': '100.0'}
```

```
>>> time_step_with_get = parser.get('simulation', 'time_step')
>>> print(time_step_with_get)
0.01
```

```
>>> time_step = parser['simulation']['time_step']
>>> print(time_step)
0.01
```

Access a value of a keyword in a section

By default everything is a str but You can use *getint(), getfloat() or getboolean()*

Don’t touch your code anymore!

In some occasion, you might want to write configuration files as well!

It starts like before:

```
#Let's import the ConfigParser object directly
from configparser import ConfigParser

# And create a config object
config = ConfigParser()
```

Don’t touch your code anymore!

Creating a configuration is equivalent to create dictionaries:

Section

Content of section

```
config['simulation'] = {'time_step': 1.0, 'total_time': 200.0}
config['environment'] = {'gravity': 9.81, 'air_resistance': 0.02}
config['initial_conditions'] = {'velocity': 5.0, 'angle': 30.0, 'height': 0.5}
```

Don’t touch your code anymore!

And then you save it into a file:

Open in ‘write’ mode

Name of the final file

```
with open('config_file_program.ini', 'w') as configfile:
    config.write(configfile)
```

That’s what you get:

```
[simulation]
time_step = 1.0
total_time = 200.0

[environment]
gravity = 9.81
air_resistance = 0.02

[initial_conditions]
velocity = 5.0
angle = 30.0
height = 0.5
```

Don’t touch your code anymore!

Conf and CLI challenge #2

Consider the following INI file:

```
[fruits]
oranges = 3
lemons = 6
apples = 5

[vegetables]
onions = 1
asparagus = 2
beetroots = 4
```

-Read it using the configparser library

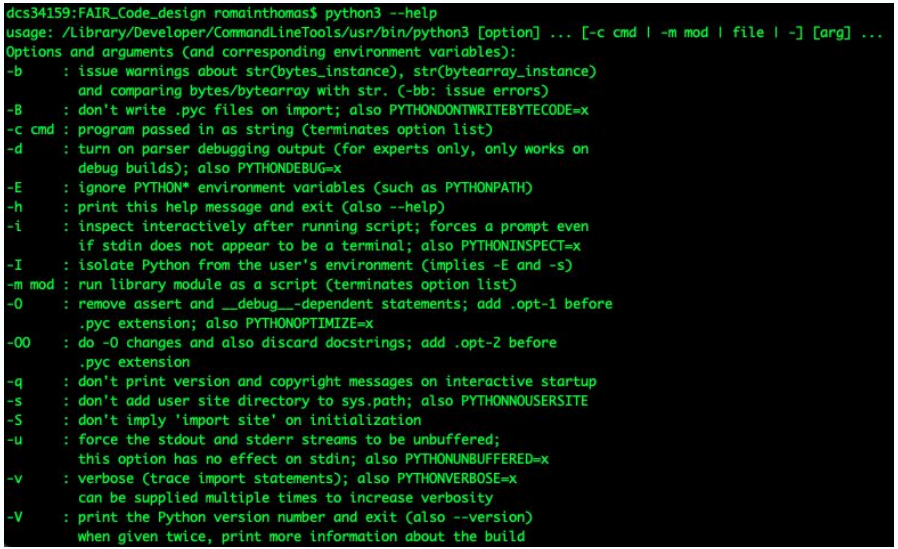
-change the number of beetroots to 2 and the number of oranges to 5.

-Add a section ‘pastries’ with 5 croissants.

-Then save it back on disk in a different file.

Don’t touch your code anymore!

Command line interfaces [CLI] is a text based interface used to interact with software and operating systems.



Don’t touch your code anymore!

Configuration: CLI allows you to modify the configuration without touching the code.

Quick execution: It is often faster to use a CLI then using a GUI

Documentation: CLI helps document the use of your code (--help command)

Adding new feature: Adding arguments is very easy

HPC and batch processing: HPC are often giving you a terminal. Using CLI makes the use of your code easy. CLI commands can be easily scripted to automate tasks.

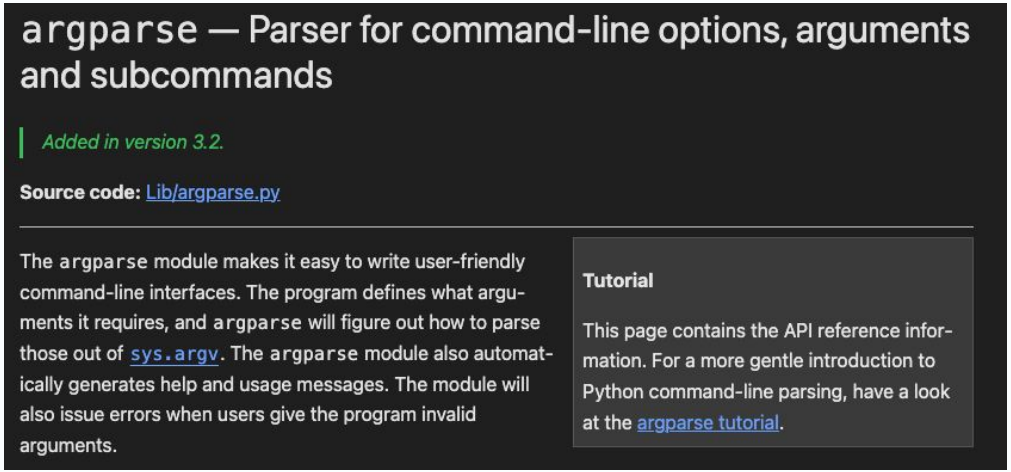
```
dcs34159:FAIR_Code_design romainthomas$ python3 --help
usage: /Library/Developer/CommandLineTools/usr/bin/python3 [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-b     : issue warnings about str(bytes_instance), str(bytearray_instance)
         and comparing bytes/bytearray with str. (-bb: issue errors)
-B     : don't write .pyc files on import; also PYTHONDONTWRITEBYTECODE=x
-c cmd : program passed in as string (terminates option list)
-d     : turn on parser debugging output (for experts only, only works on
         debug builds); also PYTHONDEBUG=x
-E     : ignore PYTHON* environment variables (such as PYTHONPATH)
-h     : print this help message and exit (also --help)
-i     : inspect interactively after running script; forces a prompt even
         if stdin does not appear to be a terminal; also PYTHONINSPECT=x
-I     : isolate Python from the user's environment (implies -E and -s)
-m mod : run library module as a script (terminates option list)
-O     : remove assert and __debug__-dependent statements; add .opt-1 before
         .pyc extension; also PYTHONOPTIMIZE=x
-OO    : do -O changes and also discard docstrings; add .opt-2 before
         .pyc extension
-q     : don't print version and copyright messages on interactive startup
-s     : don't add user site directory to sys.path; also PYTHONNOUSERSITE
-S     : don't imply 'import site' on initialization
-u     : force the stdout and stderr streams to be unbuffered;
         this option has no effect on stdin; also PYTHONUNBUFFERED=x
-v     : verbose (trace import statements); also PYTHONVERBOSE=x
         can be supplied multiple times to increase verbosity
-V     : print the Python version number and exit (also --version)
         when given twice, print more information about the build
```

Don’t touch your code anymore!

Creating a command line interface in Python

Python gives you the *argparse* library. Allows you to create user friendly command line interfaces with a very limited amount of line.

→ Part of the standard library.



`argparse` — Parser for command-line options, arguments and subcommands

Added in version 3.2.

Source code: Lib/argparse.py

The `argparse` module makes it easy to write user-friendly command-line interfaces. The program defines what arguments it requires, and `argparse` will figure out how to parse those out of `sys.argv`. The `argparse` module also automatically generates help and usage messages. The module will also issue errors when users give the program invalid arguments.

Tutorial

This page contains the API reference information. For a more gentle introduction to Python command-line parsing, have a look at the argparse tutorial.

Don’t touch your code anymore!

Let’s create one!

No need to install it!

This is seen by the user when asking for the help

```
###import the library
import argparse


###create the parser object
parser = argparse.ArgumentParser(description='This program is an example of command line interface in Python',
                                 epilog='Author: R. Thomas, 2024, UoS')
```

Then you need to tell your code to analyse (parse) the arguments passed by the user:

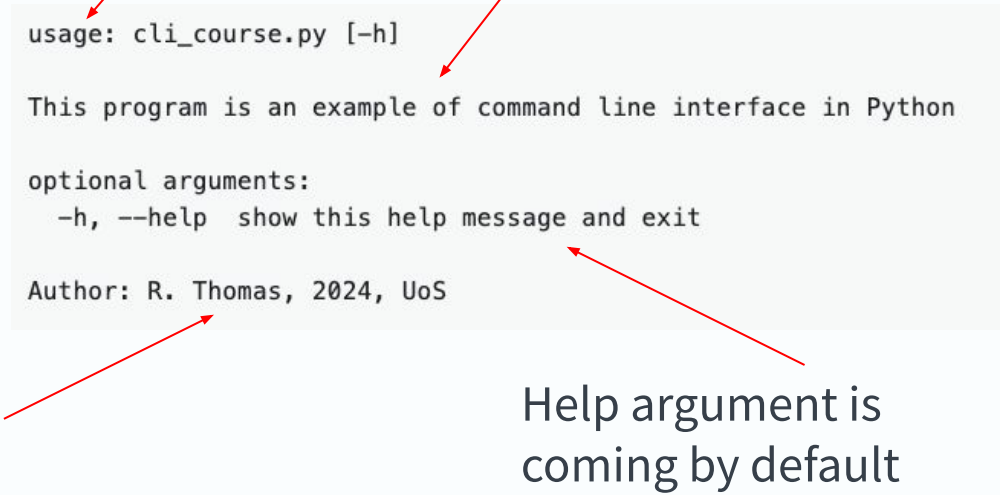
```
args = parser.parse_args()
```

Don’t touch your code anymore!

You just created your first CLI!

Save the previous lines in a file called *‘cli_course.py’ and run:*

```
python3 cli_course.py --help
```



Don’t touch your code anymore!

You just created your first CLI! But it is not very useful for the moment.

Let’s add arguments!

Two types exist:

- *Optional arguments*: start by `-` or `--` and are called in the terminal by their name. They **can be ignored** by the user.

- *Positional arguments:* Their name DO NOT start by `-` or `--`, the user **cannot ignore them** and they are not to be called by their name (just the value needs to be passed).

Don’t touch your code anymore!

You just created your first CLI! But it is not very useful for the moment.

Let’s add arguments!

In practice (before the line *args = parser.parse_args()*):

```
parser.add_argument('file')                 # positional argument (mandatory)
parser.add_argument('file2')                # positional argument (mandatory)
parser.add_argument('-c', '--count')        # option that takes a value
parser.add_argument('-n')                   # option that takes a value
parser.add_argument('--max')                # option that takes a value
```

Don’t touch your code anymore!

You just created your first CLI! But it is not very useful for the moment.

Let’s add arguments!

```
usage: cli_course.py [-h] [-c COUNT] [-n N] [--max MAX] file file2

This program is an example of command line interface in Python

positional arguments:
  file
  file2

optional arguments:
  -h, --help            show this help message and exit
  -c COUNT, --count COUNT
  -n N
  --max MAX

Author: R. Thomas, 2024, UoS
```

Mandatory arguments

Non-Mandatory arguments

Don’t touch your code anymore!

You just created your first CLI! But it is not very useful for the moment.

Let’s add arguments!

Like this, the help is not very helpful. We don’t know what the argument are there for. → Help the user!

Description for each argument

```
parser.add_argument('file', help='input data file to the program')          # position argument
parser.add_argument('file2', help='Configuration file to the program')      # position argument
parser.add_argument('-c', '--count', help='Number of counts per iteration') # option that takes a value
parser.add_argument('-n', help='Number of iteration')                       # option that takes a value
parser.add_argument('--max', help='Maximum population per iteration')       # option that takes a value
```

Don’t touch your code anymore!

You just created your first CLI! But it is not very useful for the moment.

Let’s add arguments!

For the moment, the user can pass whatever values and they would be accepted.
It is possible to tune the argument a little bit more:

‘actions’: allows you to store boolean values

→ ex: *parser.add_argument(‘-x’, **action=’store_true’**)*

‘default’: allows you to add a default value to the argument in case the user does not use it.

→ ex: *parser.add_argument(‘-x’, **default=5**)*

Don’t touch your code anymore!

You just created your first CLI! But it is not very useful for the moment.

Let’s add arguments!

For the moment, the user can pass whatever values and they would be accepted.
It is possible to tune the argument a little bit more:

‘type’: enforce the type of value that the user must give

→ ex: *parser.add_argument(‘-x’, **type=int**)*

‘choices’: Enforce the value to be passed to be amongst a list of allowed values

→ ex: *parser.add_argument(‘-x’, **choices=[‘blue’, ‘red’, ‘green’]**)*

Don’t touch your code anymore!

You just created your first CLI! But it is not very useful for the moment.

Now we must get the values from the CLI!
We already know that line:


```
args = parser.parse_args()
```

Everything is in ‘args’:

```
args = parser.parse_args()
print(args) # Gives the namespace content
print(args.file) #direct access to the 1st positional argument
print(args.max) #direct access to the max optional argument
```

Don’t touch your code anymore!

You just created your first CLI! But it is not very useful for the moment.

Error!!!!

```
[user@user]$ python cli_course.py file1path/file.py -c 3 --max 5
usage: cli_course.py [-h] [-c COUNT] [-n N] [--max MAX] file file2
cli_course.py: error: too few arguments  ####<---One positional argument is missing.

[user@user]$ python cli_course.py file1path/file.py file2path/file2.py -c 3
Namespace(count='3', file='file1path/file.py', file2='file2path/file2.py', max=None, n=None)
file1path/file.py
None
```

1st positional arguments

All of them!

‘max’

Don’t touch your code anymore!

Conf and CLI challenge #3

Create a Python script called `basic_cli.py` that:

Accepts two arguments: `--input_file` (path to the data file) and `--output_dir` (directory for saving results). Prints out the values of these arguments.

Expected output:

```
Input file: /data/input.txt
Output directory: /results/
```

Don’t touch your code anymore!

Final challenge!

Check your mails!

Help us improve!

Scan to give us feedback!