Dikes For Dummies

A Python Object Oriented course.

Carles S. Soriano Perez

Copyright © 2022 Carsopre

Table of contents

Copyright © 2022 Carsopre

1. Dikes For Dummies

Dikes for dummies is a "fast" Python course that navigates the creation of a Minimal Viable Product (MVP) starting by the project creation until document publishment while having a look on Object Oriented principles and the building of your own test suite to increase your code quality.

1.1 Requirements.

For simplicity reasons we recommend having the following installed in your computer:

- Anaconda latest version (Python >= 3.9).
- It is suggested to include it in the system's Path to operate fully through console.
- Visual Studio Code. (Easy to install through Anaconda suite).
- Other IDEs like PyCharm are totally acceptable, however the debugging steps described in the chapters are aimed for VSCode.

1.2 Following from a chapter.

During the walk-through the chapters, we will see snippets of code that can help the reader build on their own a solution that satisfies the study case. Sometimes the snippets will not be enough, this is intended. However, if the reader gets lost they can check out the corresponding branch for each chapter.

To 'take-on' from any chapter branch, you just need to do the following:

```
conda env create -f environment.yml
conda activate dikes-for-dummies_env
poetry install
```


Copyright © 2022 Carsopre

2. Study Case

We want to create a tool capable of calculating the geometry for a dike reinforcement.

A dike reinforcement is a series of operations that take place on a previously defined dike. These operations will afect its structure, and therefore its geometry.

The user will provide the following input:

- Dike profile characteristic points (8 points, 4 polderside, 4 waterside).
- Reinforcement data:
- Height
- Width
- Cost of material.

The user wants to know:

- The costs for all possible reinforcements with the data given.
- The geometries for all possible reinforcements with the data given.

2.1 Minimal functional requirements.

- The tool can be used a python library.
- The tool outputs the data to a directory given by the user.

2.2 Minimal non-functional requirements.

- The tool has 80% code coverage.
- The tool has no failing tests.
- The public methods are documented.

2.3 Extra

- The tool accepts an ini file (or other) containing all required input data.
- The tool generates all output in one file.
- The tool has a CLI interface.
- The tool has a GUI interface.
- The tool can be run as a stand-alone exe.
- Both minimal functional and non-fuctional requirements should still be met.

Copyright © 2022 Carsopre

3. Chapters

3.1 Chapter 01. Project setup

In this Chapter we will be setting up the project and our VScode settings.

3.1.1 IDE's and recommended plugins.

Setting up the project starts by selecting an IDE. We will use VSCode out of personal satisfactory experience and the well integration of pluggins that help on debugging or code maintainance tasks. The first thing to do with VSCode is to create our own configuration for the project. You can either do this via the interface or my preferred method by creating a `settings.json` file in the `.vscode` directory located in the root of the project such as follows:

```
{
    "terminal.integrated.profiles.windows": {
        "conda": {
            "path": "C:\\Windows\\System32\\cmd.exe",
            "args": [
                "/K",
                "C:\\Anaconda3\\Scripts\\activate.bat",
                "C:\\Anaconda3"
            ]
        }
    },
    "terminal.integrated.defaultProfile.windows": "Command Prompt",
    "terminal.integrated.cwd": "${workspaceFolder}",
    "editor.minimap.enabled": false,
    "editor.formatOnSave": true,
    "python.formatting.provider": "black",
    "python.defaultInterpreterPath": "C:\\Anaconda3\\envs\\dikes-for-dummies_env\\python.exe",
    "python.terminal.activateEnvironment": true,
    "python.linting.mypyEnabled": false,
    "python.linting.enabled": true,
    "python.testing.cwd": "${workspaceFolder}",
    "python.testing.unittestEnabled": false,
    "python.testing.pytestEnabled": true,
    "python.testing.pytestArgs": [
        ""
    ],
    "autoDocstring.startOnNewLine": true,
    "autoDocstring.docstringFormat": "google",
    "[python]": {
        "editor.formatOnSave": true,
        "editor.codeActionsOnSave": {
            "source.organizeImports": true
        }
    },
    "python.sortImports.args": [
        "--profile=black"
    ],
}
```

As can be seen, we are declaring already the usage of certain python libraries such as `black` and `pytest`, plugins from VSCode such as `autoDocstring`, and an anaconda environment to contain our working environment dependencies on it. It is also advised to include the `.vscode` directory in the .gitignore file.

Some recommended plugins (extensions) I usually have always on in VSCode:

- Python. Basically the only plugging that is non-officially **required** to code in Python. It will help you with linting, debugging, code-formatting, refactor, etc.
- autoDocstring. A **must** for automatically generating docstring headers anywhere in your code.
- Lorem ipsum. Very handy to generate texts for testing purposes.
- Test Explorer UI Better test ordering and in-code runner.

Of course there are many more, don't be shy and explore whatever fits best your needs.

Copyright © 2022 Carsopre

3.1.2 Adding dependencies.

One of the biggest issues in python packaging is keeping all the dependencies derived from your packages in order as some of them have strong version requirements with third party libraries.

To deal with this problem you can either always keep an eye on this yourself or instead take advantage of package handlers such as poetry. Poetry is capable of installing all of your required dependencies while keeping them to the optimal version between themselves. All its work will be usually declared in a `pyproject.toml` file that can be modified via poetry commands or manually.

However, poetry won't be able to install a package if this requires certain `wheels`. In short, a `wheel` is a pre-compiled library that cannot be natively installed in your system.

A good example is GDAL. To solve this problem the easiest solution when working on a Windows machine is to have a `conda environment` running on the background. Anaconda can install for us mentioned wheel which will allow poetry to add it to our package without further conflicts. Example:

```
conda install -c conda-forge gdal=3.0.2
poetry add gdal
```

An example of a conda environment that requires wheels can be as follows:

```yaml
name: projectwithwheels_env
channels:
  - defaults
  - conda-forge
dependencies:
  - conda-forge::python>=3.8
  - conda-forge::rasterio
  - conda-forge::gdal
  - conda-forge::geos
  - conda-forge::fiona
  - pip
  - pip:
    - pytest
    - pytest-cov
    - teamcity-messages
    - poetry
```

Let's install the provided environment.yml file and start our poetry project.

```
conda install env -f environment.yml
conda activate dikes-for-dummies_env
poetry init
```

When running `poetry init` the console will start an interactive mode in which we will be able to add dependencies both for packaging but also for development (think of code formatting libraries, testing or documentation libraries for the latter). Feel free to explore it a bit, the next step will contain a stable `*.toml` file.

Once we are done with adding dependencies we will create the `src` directory under a recognisable name (but can be src as well) in it we will add our first `__init__.py` file with the current version of the tool:

```
__version__ = "0.1.0"
```

This can be also done with commitizen. This tool helps us versioning the project and keeping a neat changelog.

Now we can install our package to start working on it

```
poetry install
```

3.1.3 Documentation.

There are as many documentation packages as you may imagine. But of course you want to use the most popular ones to ensure easy 'troubleshooting' when problems occur. Some commonly used are:

- Sphinx. Easy to deploy and quite broadly used in pypi packages.
- Mkdocs. Easy to use as it follows very clear structures and documentation is written with markdown language.

Copyright © 2022 Carsopre

Most of them are capable of reading your code docstrings and generating such technical documentation for developers. So in the end you should pick the one that adapts best to your workflows and knowledge. At the same time, you may chose to generate documentation and publish it in pages such as Read the docs or GitHub Pages.

3.1.4 Publishing / Delivering.

Last, the ultimate goal of creating a product is to deliver it. We have several options:

- Create a pypi package. Users can import our package through `pip` or `poetry`. Building the package and pushing it to pypi it's relatively easy with `poetry`.
- Create an .exe. With the help of packages such as `pyinstaller` we can easily achieve this.

All of the above are, however, not exclusive. We need to consider that our package will have different audiences that will use it in different manners:

- as sandbox (think of developers extending the project)
- as a library (pypi package or directly from GitHub).
- as an endpoint tool (think of an exe with limited workflows):
- CLI
- API
- GUI
- Web server

Copyright © 2022 Carsopre

3.2 Chapter 02. Never trust the user.

Why you should never trust the user and their given input Unlike other OO languages, Python has `duck typing` and is a dynamic language. This means that even if you declared in a method that you want an object of type `Banana` you may end up getting `Peanuts` instead. For this reason, I always advice **never** to trust the user.

However, as seen in the previous chapter, we have different ways of distributing the package. We will see what this means by the end of the chapter.

We are going to start debugging some snippets in this chapter to get a grip of some python concepts, so, let's add a `launch.json` to our `.vscode` directory.

3.2.1 Methods in Python

These are ways of declaring methods and handling their input / output. Try them out.

```
def get_dike_profile_points():
    return None
get_dike_profile_points()
```

```
def get_dike_profile_points(*args, **kwargs):
    return (1,2)
x = get_dike_profile_points(abc=(3,4,5))
x1, x2 = x
```

```
def get_dike_profile_points(abc, *args, **kwargs):
    return ((1,2), (3,4), (5,6))
p1, p2 , p3= get_dike_profile_points(3,4,5)
```

As can be seen, we can collect the parameters declaring them explicetly or by using the parameters `*args` or `**kwargs`.

3.2.2 Classes in Python

Official reference.

Creating classes is very simple, you just need to add the `class` type at its begining:

```
class DikeProfile:
    pass
```

We can also apply (multiple) inheritance.

```
class SoilReinforcementProfile(DikeProfile):
    pass
```

Initialization.

We initializing classes usually with the `def __init__(self, *args, **kwargs)` method. However we can take different approaches.

- Default constructor:

```
class DikeProfile:
    def __init__(self, *args, **kwargs):
        self.name = "A default Dike Profile"
_dike_profile = DikeProfile()
```

- From a class method (later further explained):

```
class DikeProfile:
    @classmethod
    def from_data_dict(cls, **kwargs):
        _dike_profile = cls()
        _dike_profile.name = kwargs.get("name", "A default Dike Profile")
```


Copyright © 2022 Carsopre

```
        return _dike_profile
_dike_profile = DikeProfile.from_data_dict(name="Or not so default.")
```

- From a 'builder' (or a *FactoryPattern*):

```
 class DikeProfileBuilder:
    def build(self):
        _profile = DikeProfile()
        _profile.name = "A default Dike Profile"
        return _profile
```

There are certain libraries that help initializing classes while enforcing its typing. Have a look at Pydantic. Keep in mind that when using such libraries you are constrained to their potential bugs / limitations. So you should be responsible on to how to use it (later discussed in the summary).

Static methods.

These are examples of static methods:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

# outside a class
def get_tuple_as_point( x, y):
    return Point(x, y)

_point = get_tuple_as_point(4, 2)
# In a class with decorator staticmethod
class DikeProfileBuilder:
    @staticmethod
    def get_point_list(*args, **kwargs):
        _point_list = []
        for point_tuple in args:
            x, y = point_tuple
            _point_list.append(get_tuple_as_point(x, y))
        return _point_list
_point_list = DikeProfileBuilder.get_point_list((4, 2), (2,4))
```

Class methods.

A class method is similar to a static one. However, despite being technically possible to make them behave identically, a class method is meant to return an instance of the class being invoked.

```
# Do not:
class DikeProfile:
    @classmethod
    def get_point_list(cls, *args, **kwargs):
        pass
# Do:
def get_point_list(*args):
    pass
class DikeProfile:
    @classmethod
    def from_data(cls, *args, **kwargs):
        _dike_profile = cls()
        _dike_profile.points = get_point_list(args)
        return _dike_profile
_profile = DikeProfile.from_data((1,2), (3,4))
```

Overloading methods.

As in other OO languages, we can also overload methods:

```
class DikeProfile:
    def __str__(self):
        return "A default DikeProfile"
print(str(DikeProfile()))
```

And of course, we can also overload operators:

```
class DikeProfile:
    def __eq__(self, compare_to):
        if len(compare_to.points) != len(self.points):
            return False
        for idx, point in enumerate(self.points):
```


Copyright © 2022 Carsopre

```
            to_point = compare_to.pints[idx]
            if to_point.x != point.x or to_point.y != point.y:
                return False
        return True

_dike_a = DikeProfile()
_dike_a.points = [Point(1, 2), Point(2, 3)]
_dike_b = DikeProfile()
_dike_b.points = [Point(1, 2), Point(2, 3)]
assert _dike_a == _dike_b
_dike_b.points = [Point(1, 2), Point(2, 4)]
assert _dike_a != _dike_b
```

Handling errors:

Some examples on how to handle errors in Python:

```
try:
    _result = 1 / 0
except Exception as exc_info:
    ...
finally:
    ...
```

```
from pathlib import Path
def read_file(file):
    if not file:
        raise ValueError("File not provided")
    if not isinstance(file, Path):
        file = Path(file)
    if not file.is_file():
        # This error is futile as the built-in read_text would raise it anyway.
        raise FileNotFoundError(f"File not found at {file}")
    _lines = file.read_text().splitlines(keepends=False)
```

3.2.3 Clean code (or the most effective ways to make it clean).

This part ends up being more a responsibility than a mechanism we can build (except for certain code formatters.) The first advice, is to adhere to the following: - Dry code. - Single responsibility principle (OO). - Classes and methods standardization. - Use code formatters and related tools, the most simple one `black`.

```
poetry run black .
```

- You can also order your imports with `isort`.

```
poetry run isort .
```

- Descriptive (consistent) variables.

```
# Don't
_mcf = a_calculation_that_happens_somewhere(a,b)
# Do
_geometry_area = calculate_geometry(_list_of_points)
```

- Documented code. - Type hinting

```
from __future__ import annotations

from typing import List, Tuple


class Point:
    x: float
    y: float

    def __init__(self, x: float, y: float) -> None:
        self.x = x
        self.y = y

    def __str__(self) -> str:
        return f"Point ({self.x}, {self.y})"


class DikeProfile:
    def __init__(self):
        self._points = []

    @classmethod
    def with_data(cls, point_tuples: List[Tuple[float]]) -> DikeProfile:
        _profile = cls()
```


Copyright © 2022 Carsopre

```
        _profile.points = [Point(*_pt) for _pt in point_tuples]
        return _profile

    @property
    def points(self) -> List[Point]:
        return self._points

    @points.setter
    def points(self, list_values: List[Point]):
        self._points = list_values


_dike = DikeProfile.with_data([(1, 2), (2, 3)])
for point in _dike.points:
    print(point)
```

- Docstrings (pick your preferred format and be **consistent** about it). When using autoDocstring, you only need to write three " and press enter to generate a template.

```
class DikeProfile:
    pass
class DikeReinforcementInput:
    pass
class ReinforcedDikeProfile(DikeProfile):
    pass

def profile_calculator(profile: DikeProfile, new_input: DikeReinforcementInput) -> ReinforcedDikeProfile:
    """
    Calculates a new profile based on the given `profile` and `new_input`.

    Args:
        profile (DikeProfile): Base profile on which calculations will be done.
        new_input (DikeReinforcementInput): Data input required to perform calculations.

    Returns:
        ReinforcedDikeProfile: Instance of new reinforced Dike
    """
    pass
```

3.2.4 Summary

In this chapter we have seen that in Python we can still code in an effective Object Oriented way.

However, as mentioned, because of Python being a dynamic language, and the ways of distributing the repository, we need to consider also how to handle potential errors in the code and / or users' input. Let's analyze the options:

1. As a package library (pip) or sandbox.

a. The user is an 'expert' or a 'developer'.

b. The code contains type-hints and tests ensuring the correct functioning.

c. We only try-catch on high end operations or as a part of a wrapper to a third-party package.

2. As an endpoint, CLI, API or .exe product.

a. The user does not necessarily need to know how to code.

b. The user only uses the API endpoints / calls which are well-documented.

In 1.1 the user should be responsible enough to know how to handle the package. It should be accepted to just follow a **fail-fast** philosophy in our code. However, in the second case, we could do error handling at a higher level (classic main.py) yet leaving the rest of the code intentionally following a fail-fast philosophy.

Copyright © 2022 Carsopre

3.3 Chapter 03. Objected Oriented Programming in Python

We have already seen the perks and pitfalls of Python as a dynamic language and its `duck typing`. Now we will explore both object oriented *foundations* and *solid* concepts applied through Python programming.

3.3.1 Foundations

- Encapsulation.
- Define public, protected and private data.
- Public data can be accessed 'from the outside'.
- Protected data only internally (when using inheritance).
- Private data only in the declared class.
- Inheritance and abstractions:
- Definition of generic functionality and properties in the base class.
- Concrete methods in the inherited classes.
- Abstract classes need a concrete inherited class (specialisation).
- A class can inherit from an abstract class or another concrete class.
- Polymorphism.
- The same interface applies for different data types or classes
- Can be applied to classes (through inheritance) and methods.
- Aggregation and composition.
- **Aggregation**: the associated objects do not need each other 'to exist'.
- **Composition**: the associated objects 'need' each other to 'coexist'. The main object owns

Encapsulation

In python encapsulation does not really exist, think of it more like a 'rule of conduct'. We identify protected (internal) methods and parameters with a simple underscore `_` and private with double `__` so: - `def _get_my_variable(...) -> Any` is meant to be used only while developing in the tool. Preferrebly within an instanced object. - `def __get_my_variable(...) -> Any` is only meant to be used within its class / module. And not to be exposed.

```
class BasicEncapsulation:
    def __init__(self):
        self.public_property = 42
        self._protected_property = 4.2
        self.__private_property = 0.42

    def public_method(self):
        pass

    def _protected_method(self):
        pass

    def __private_method(self):
        pass
```

Inheritance

We have already shown simple inheritance, but of course Python allows us to do multiple inheritance whenever needed.

```
class DikeProfile:

    characteristic_points: List[Point]

    def __init__(self) -> None:
        self.characteristic_points = []

class DikeReinforcement:
```


Copyright © 2022 Carsopre

```
    reinforcement_input: DikeReinforcementInput

    def __init__(self) -> None:
        self.reinforcement_input = None

class DikeReinforcementProfile(DikeProfile, DikeReinforcement):
    def __str__(self) -> str:
        return "Reinforced Profile"


_drp = DikeReinforcementProfile()
assert isinstance(_drp, DikeReinforcement)
assert isinstance(_drp, DikeProfile)
assert isinstance(_drp, DikeReinforcement)
```

Abstractions

To create abstract methods or classes we use the library `ABC`:

```
from abc import ABC, abstractmethod

class DikeProfileBase(ABC):
    characteristic_points: List[Point]

    def __init__(self) -> None:
        self.characteristic_points = []

    @abstractmethod
    def __str__(self) -> str:
        raise NotImplementedError("Implement in concrete class")

    def set_points_from_tuples(self, tuple_list: List[Tuple[float, float]]):
        if not tuple_list:
            raise ValueError("tuple_list argument required.")
        self.characteristic_points = list(map(Point, tuple_list))

class DikeProfile(DikeProfileBase):
    ...
    def __str__(self) -> str:
        return "Initial Dike Profile"

    @classmethod
    def from_tuple_list(cls, tuple_list: List[Tuple[float, float]]) -> DikeProfile:
        _dike = cls()
        _dike.set_points_from_tuples(tuple_list)
        return _dike

assert issubclass(DikeProfile, DikeProfileBase)
```

Polymorphism

Although it's possible to apply polymorphism in Python. In my experience is seldom used. Concrete methods and well-applied SRP provide better code.

```
class DikeProfileCalculator:
    def calculate_characteristic_points(self):
        # Base dike profile operations.
        pass
class PipingDikeProfileCalculator(DikeProfileCalculator):
    def calculate_characteristic_points(self):
        # Reinforcement piping dike operations.
        return super().calculate_characteristic_point
```

3.3.2 SOLID

Single responsibility principle.

By creating 'builders' and leaving the classes only as datastructures we reduce the amount of responsibility a class needs to do.

Copyright © 2022 Carsopre

Open for extension, closed for modification.

A class should be extendable without modifying the class itself. Whenever you start having an `if-else` to differenciate behaviors, try to create a new concrete class.

- Given:

```
class DikeMaterial:
    def __init__(self, cost: float, material_type: str):
        self.cost = cost
        self.material_type = material_type

    def get_material_cost(self) -> float:
        if self.material_type == "sand":
            return self.cost
        else:
            return self.cost * 1.5

def get_total_cost(self, material_list: List[DikeMaterial]) -> float:
    return max(_material.get_material_cost() for _material in material_list)

_material_list = [DikeMaterial(2.4, "sand"), DikeMaterial(4.2, "clay")]
get_total_cost(_material_list)
```

- We can do instead:

```
class DikeMaterial:
    def __init__(self, price: float):
        self.price = price

    def get_material_cost(self) -> float:
        return self.price

class ClayMaterial(DikeMaterial):
    def get_material_cost(self):
        return self.price * 1.5

_material_list = [Bridge(2.4), ClayMaterial(4.2)]
_costs = sum(_m.get_material_cost() for _m in material_list)
```

In addition, initaiting classes through *classmethods*, or external constructors / factories might allow you to easily create one or the other. Delegating even more responsibilities and being more aligned with the *Open-closed* principle. Example:

```
from typing import Protocol
from __future__ import annotations

class MaterialProtocol(Protocol):
    price: float
    name: str

    def get_cost(self) -> float:
        pass

class SandMaterial(MaterialProtocol):
    price: float

    def get_cost(self) -> float:
        return self.price * 1.14

    @classmethod
    def initiate_with_price(cls, price: float) -> Bridge:
        # Through class methods.
        _material = cls()
        _material.price = price
        return _material

def build_material(material_type: Type[MaterialProtocol], price: float) -> MaterialProtocol:
    # Through a 'builder'
    _material = bridge_type()
    _material.price = price
    return _material
```

Liskov substitution.

A subclass must be substitutable by its super class. In my opinion, this principle is not really applyable in Python. Examples below:

```
class SuperSand(SandMaterial):
    def get_cost(self) -> float:
        # Super taxed!
        return self.price * 2
```


Copyright © 2022 Carsopre

```
_material = SuperSand.initiate_with_price(2)
# 1. First we prove the bridge is the same type as the base.
assert isinstance(_material, SandMaterial), "Failed principle!"

# 2. Then we prove the cost will be the same regardless of how it is typed (parent or subclass)
def get_as_sand_material(s_material: SandMaterial) -> float:
    return s_material.get_cost()

# To make it work, we should invoke 'super':
_m_cost = _material.get_cost()
assert get_as_sand_material(super(SuperSand, _material)) != _m_cost, "Failed principle!"
assert get_as_sand_material(_material) != _m_cost, "Failed principle!"
```

Interface segregation principle.

Intefaces in Python are relatively "new", you can implement interfaces through `typing.Protocol`.

```
from typing import List, Protocol, Tuple
from shapely.geometry import Point
from typing_extensions import runtime_checkable

@runtime_checkable
class DikeProfileProtocol(Protocol):
    characteristic_points: List[Point]
    height: float
    width: float

class DikeProfile(DikeProfileProtocol):
    ...

_dike = DikeProfile()
assert isinstance(_dike, DikeProfileProtocol)
# A protocol can't be instantiated, try this:
DikeProfileProtocol()
```

Notice that the properties can be defined as inlines in the protocol. As long as they are declared later on (either with decorators or as inlines) the contract will be fulfilled.

Pay attention as well to `@runtime_checkable`, it will allow us to verify whether an instance implements said protocol.

Dependency inversion principle.

Depend on abstractions, not concretions.

```
from matplotlib import pyplot
from shapely.geometry import LineString
from dikesfordummies.dike.dike_profile import DikeProfile

def _plot_line(ax, ob, color):
    parts = hasattr(ob, "geoms") and ob or [ob]
    for part in parts:
        x, y = part.xy
        ax.plot(x, y, color=color, linewidth=3, solid_capstyle="round", zorder=1)


def plot_profile(dike_profile: ReinforcementDikeProfile) -> pyplot:
    fig = pyplot.figure(1, dpi=90)
    _subplot = fig.add_subplot(221)
    _plot_line(
        _subplot, LineString(dike_profile.characteristic_points), color="#03a9fc"
    )
    return fig
```

In theory, the above code should only work for a ReinforcementDikeProfile, well, it's python so it will swallow also a regular profile. But we should aim to make the methods / classes depending on higher level of abstractions, so we could replace it with either the base class, or a protocol.

```
def plot_profile(dike_profile: DikeProfile)
...
def plot_profile(dike_profile: DikeProfileProtocol)
```


Copyright © 2022 Carsopre

3.4 Chapter 04. The pythonic way.

Because we are doing OO in python it is also useful to learn certain Python techniques that can be handy specially when dealing with large amounts of data.

-- "Ask for forgiveness, not for permission" --

Let's '*pythonize*' the following snippet.

```
from typing import List, Tuple
from shapely.geometry import Point

def get_as_points_if_even(tuple_points: List[Tuple[float]]) -> List[Point]:
    def is_even(number: float) -> bool:
        return number % 2 == 0

    _list = []
    for p_tuple in tuple_points:
        if is_even(p_tuple[0]) and is_even(p_tuple[1]):
            _list.append(Point(p_tuple))
    return _list

_points = get_as_points_if_even([(1, 1), (2, 4), (3, 7), (4, 10)])
```

3.4.1 List comprehensions.

```
def get_as_points_if_even(tuple_points: List[Tuple[float]]) -> List[Point]:
    def is_even_tuple(p_tuple: Tuple[float]) -> bool:
        return all(_p % 2 == 0 for _p in p_tuple)

    return [Point(p_tuple) for p_tuple in tuple_points if is_even_tuple(p_tuple)]
```

3.4.2 Filter.

Returns an iterable when a condition is met.:

```
def get_as_points_if_even(tuple_points: List[Tuple[float]]) -> List[Point]:
    def is_even_tuple(p_tuple: Tuple[float]) -> bool:
        return all(_p % 2 == 0 for _p in p_tuple)

    return [Point(p_tuple) for p_tuple in filter(tuple_points)]
```

3.4.3 Map.

Returns an iterable of a callable applied in a collection of items:

```
def get_as_points_if_even(tuple_points: List[Tuple[float]]) -> Iterable[Point]:
    def is_even_tuple(p_tuple: Tuple[float]) -> bool:
        return all(_p % 2 == 0 for _p in p_tuple)

    return map(Point, filter(is_even_tuple, tuple_points))

_points = list(get_as_points_if_even([(1, 1), (2, 4), (3, 7), (4, 10)]))
```

3.4.4 Zip.

Returns an iterable of paired tuples:

```
_coords = range(1, 12)
_x_points = _coords[:4]
_y_points = _coords[0::3]
_points = list(get_as_points_if_even(zip(_x_points, _y_points)))
```

3.4.5 Lambda functions.

A lambda expression is nothing else than a 'inline method', we can use it in combination of the above generators / iterables.

Copyright © 2022 Carsopre

```
_points: Iterator[Point] = get_as_points_if_even(zip(_x_points, _y_points))
# We are going to increment the y coordinates by two values:
_new_points = list(map(lambda _p: Point(_p.x, _p.y + 2), _points))
```

Iterators are 'consumed'. Once you iterate over them they will no longer contain data. Check what happens when you inspect `_points` now.

```
# These asserts are equivalent under Python's eyes.
assert not _points
assert len(_points) == 0
```

3.4.6 Dictionaries.

Very helpful in many ways, you can use it to store any kind of callable, iterable or python object and iterate over the entire collection.

Some basic snippets: - Declaring a dictionary:

```
simple_dict = {"a": 123}
# or
simple_dict = dict(a=123)
# but for both is possible:
simple_dict["b"] = 456
```

- Accessing a dictionary:

```
assert simple_dict.keys() == ["a", "b"]
assert simple_dict.values() == [123, 456]
assert simple_dict.items() == [("a", 123), ("b", 456)]
# this will raise:
_c_value = simple_dict["c"]
# But this don't:
_c_value = simple_dict.get("c", None)
```

- Providing the contents of a dictionary:

```
_profile_data = dict(points=_points, name="A Test Dike")
class DikeProfile:
    points: List[Point]
    name: str

    def __init__(self, points: List[Point], name: str) -> None:
        self.points = points
        self.name = name

_profile = DikeProfile(**_profile_data)
```

- Assigning the values of a dictionary directly to a class:

```
class DikeProfile:
    points: List[Point]
    name: str

    def __init__(self) -> None:
        self.points = []
        self.name = ""

    @classmethod
    def with_data_dict(cls, data_dict: dict) -> DikeProfile:
        _profile = cls()
        _profile.__dict__ = data_dict
        return _profile


_profile = DikeProfile.with_data_dict(dict(points=_points, name="A Test Dike"))
assert isinstance(_profile, DikeProfile)
```

3.4.7 Logging

It is quite common wanting to log the different steps or actions throughout your entire solution. This can easily be achieved with the library `logging`.

```
import logging

# Defining our logger.
_logger = logging.getLogger("")
_logger.setLevel(logging.DEBUG)
```


Copyright © 2022 Carsopre

```
# Defining our custom formatter
_formatter = logging.Formatter(
    fmt="%(asctime)s - [%(filename)s:%(lineno)d] - %(name)s - %(levelname)s - %(message)s",
    datefmt="%Y-%m-%d %I:%M:%S %p",
)

# Adding a console handler
_console_handler = logging.StreamHandler()
_console_handler.setLevel(logging.INFO)
_console_handler.setFormatter(_formatter)
_logger.addHandler(_console_handler)

# Adding a file handler.
_log_file = Path(__file__).parent / "dikes_for_dummies.log"
_file_handler = logging.FileHandler(filename=_log_file, mode="w")
_file_handler.setLevel(logging.INFO)
_file_handler.setFormatter(_formatter)
_logger.addHandler(_file_handler)
```

3.4.8 Disposable classes.

A nice technique when wanting to create `disposable` objects that should realize initialization and finalize operations during a very specific period can be achieved with `__enter__` and `__exit__`. A good example of this are file streams. But you may also think of your own logging (instead of default library `logging`) or your own wrappers around runners.

Example for a logger:

```
class ExternalRunnerLogging:
    ...
    def __enter__(self) -> None:
        _runner_name = self._get_runner_name()

        self._wrap_message(f"Initialized Runner Logging for {_runner_name}")

    def __exit__(self, *args, **kwargs) -> None:
        _logger = logging.getLogger("")
        _runner_name = self._get_runner_name()
        self._wrap_message(f"Logger terminated for {_runner_name}")
        _logger.removeHandler(self._file_handler)
```

And its usage:

```
def run(self):
    ...
    with ExternalRunnerLogging(self):
        ...
        try:
            ...
        except Exception as exc:
            logging.info(f"Error during runner execution {exc}")
            ...
```


Copyright © 2022 Carsopre

3.5 Chapter 05. Testing 101

We are now going to start writing tests to either verify our current implementations but also to start doing some Test Driven Development.

If you installed the package (`poetry install`) you should have already pytest, otherwise add it to your .toml like follows:

```
poetry add pytest --dev
```

3.5.1 Pre-conditions.

Pytest will automatically detect your tests under the following conditions:

- they are in the tests directory.
- test files start with test_ prefix.
- test methods start with prefix test_

3.5.2 Building the test structure.

Like in many other things in Python, there is not just one way to do this. However, my recommendation is to have a test directory under your root project, at the same level of your package, in our case it should look now like this:

```
    \dikes-for-dummies
        \docs
        \dikesfordummies
            \dike
                __init__.py
                dike_profile.py
                dike_reinforcement_input.py
                dike_reinforcement_profile.py
            __init__.py
        \tests
            __init__.py
        environment.yml
        pyproject.toml
        README.md
        LICENSE
```

As we build up tests in our package I like to 'mirror' the structure in the code directory, so it's easier to understand what is actually being covered. So something like:

```
    \dikes-for-dummies
        \docs
        \dikesfordummies
            \dike
                __init__.py
                dike_profile.py
                dike_reinforcement_input.py
                dike_reinforcement_profile.py
            __init__.py
        \tests
            \dike
                __init__.py
                test_dike_profile.py
                test_dike_reinforcement_input.py
                test_dike_reinforcement_profile.py
            __init__.py
            test_acceptance.py
        environment.yml
        pyproject.toml
        README.md
        LICENSE
```

3.5.3 Creating a test

We have pytest and a file already in the tests directory. But now we miss tests, let's write one to verify the DikeProfile class:

```
import math

from dikesfordummies.dike.dike_profile import DikeProfile
```


Copyright © 2022 Carsopre

```
def test_initiate_dikeprofile():
    _dike = DikeProfile()
    assert isinstance(_dike, DikeProfile)
    assert not _dike.characteristic_points
    assert math.isnan(_dike.height)
    assert math.isnan(_dike.width)
```

If everything is in order the test explorer should be displaying now this test to run or debug.

In case it is not being display, it is a good occasion to check the python output console and check what might be causing the error.

Because of Python not being compiled, discovering tests is (in occasions) the best way to ensure your solution is *problem free*.

It is also possible to run the tests via command line:

```
poetry run pytest -V
```

You could also encapsulate the test in a class (my preferred choice).

```
import math

from dikesfordummies.dike.dike_profile import DikeProfile

class TestDikeProfile:

    def test_initiate_dikeprofile(self):
        _dike = DikeProfile()
        assert isinstance(_dike, DikeProfile)
        assert not _dike.characteristic_points
        assert math.isnan(_dike.height)
        assert math.isnan(_dike.width)
```

Verifying risen errors:

Of course we can also test that an error is risen:

```
def test_given_no_tuple_list_when_from_tuple_list_then_raises():
    _expected_err = "tuple_list argument required."
    with pytest.raises(ValueError) as exc_err:
        DikeProfile.from_tuple_list(None)
    assert str(exc_err.value) == _expected_err
```

Adding multiple cases.

By now, you should be wondering how to apply DRY to your tests. For instance, in the previous section, we could have also given an empty list (`[]`) because python operator `not` will also consider it as if it was `None` value.

We will be using `pytest.mark.parametrize` and `pytest.param` for this:

```
@pytest.mark.parametrize(
    "list_value",
    [pytest.param(None, id="None value"), pytest.param([], id="Empty list")]
)
def test_given_no_tuple_list_when_from_tuple_list_then_raises(list_value: Any):
    _expected_err = "tuple_list argument required."
    with pytest.raises(ValueError) as exc_err:
        DikeProfile.from_tuple_list(list_value)
    assert str(exc_err.value) == _expected_err
```

Our test suite should now detect multiple test cases for this test and allow us to run them either individually or all together.

This feature allow us for many possibilities. For instance:

```
@pytest.mark.parametrize("a", [(1), (2), (3)])
@pytest.mark.parametrize("b", [(4), (5), (6)])
def test_dummy_multi_parameter(a: float, b: float):
    assert (a / b) <= 0.75
```

We will have a total of 9 cases, because all possibilities will be crossed.

You can verify exact float values with pytest.approx(expected_value, tolerance)

Copyright © 2022 Carsopre

Furthermore, pytest allows you to decide which tests to execute and whic note. for instance by using the custom decorator `@pytest.mark.acceptancetest` and running the specific command `pytest -v -m acceptancetest` we will only run the tests with said decorator.

We can also skip tests if, for instance, we do not wish to run them under certain conditions with `@pytest.mark.skipif()`:

```
import os

@pytest.mark.skipif(
    os.platform.system().lower() != "linux", reason="Only Linux supported"
)
def test_only_run_this_test_in_linux():
    pass
```

Of course, we can create our own markers to stay *DRY*:

```
import platform

only_linux = pytest.mark.skipif(
    platform.system().lower() != "linux", reason="Only Linux supported"
)

@only_linux
def test_a_test_for_linux():
    pass

@only_linux
def test_another_for_linx():
    pass
```

Using fixtures.

Pytest also allows us for tear up / tear down fixtures. These are broad, so we will just see a few common examples. - Providing a common object across the test suite.

```
@pytest.fixture(scope="function", autouse=False)
def base_dikeprofile():
    _dike = DikeProfile()
    _dike.characteristic_points = list(map(Point, zip(range(0, 4), range(0, 4))))
    assert len(_dike.characteristic_points) == 4
    assert _dike.width == _dike.height == 3


def test_using_a_fixture(base_dikeprofile: DikeProfile):
    assert isinstance(base_dikeprofile, DikeProfile)
```

- Providing initial data and cleaning it up afterwards.

```
@pytest.fixture(scope="module")
def base_output_dir():
    _test_results = Path(__file__).parent / "test_results"
    _test_suite_results = _test_results / "acceptance_tests"
    _test_suite_results.mkdir(parents=True, exist_ok=True)

    # Give data to the test.
    yield _test_suite_results

    # This code executes after the test ends
    if _test_suite_results.is_dir():
        shutil.rmtree(_test_suite_results, ignore_errors=True)


def test_using_a_fixture(base_output_dir: Path, request: pytest.FixtureRequest):
    this_test_results = base_output_dir / request.node.name
    this_test_results.mkdir(parents=True)
    pytest.fail("This is a dumb test")
```

3.5.4 Workflows on GitHub

Yoy may find multiple solutions by googling this, but our most common pipeline in Python is something similar to this:

```
name: ci-on-push-and-autoformat
on:
  pull_request:
    types: [opened, synchronize, reopened]
  push:
    branches:
      - master
jobs:
```


Copyright © 2022 Carsopre

```
  CI:
    if: "!startsWith(github.event.head_commit.message, 'bump:')"
    strategy:
      fail-fast: false
      matrix:
        python-version: ['3.10']
        os: [ubuntu-latest, windows-latest, macos-latest]
    runs-on: ${{ matrix.os }}
    permissions: write-all
    steps:
      - uses: actions/checkout@v2
        with:
          fetch-depth: 0

      - name: Set up Python
        uses: actions/setup-python@v1
        with:
          python-version: ${{ matrix.python-version }}

      - name: Run image
        uses: abatilo/actions-poetry@v2.0.0
        with:
          poetry-version: 1.1.8
      - name: Cache Poetry virtualenv
        uses: actions/cache@v1
        id: cache
        with:
          path: ~/.virtualenvs
          key: venv-${{ matrix.os }}-${{ matrix.python-version }}-${{ hashFiles('**/poetry.lock') }}
          restore-keys: |
            venv-${{ matrix.os }}-${{ matrix.python-version }}-

      - name: Set Poetry config
        run: |
          poetry config virtualenvs.in-project false
          poetry config virtualenvs.path ~/.virtualenvs

      - name: Install Dependencies
        run: poetry install
        if: steps.cache.outputs.cache-hit != 'true'

      - name: Test with pytest
        run: |
          poetry run pytest --cov=dikesfordummies --cov-report xml:coverage-reports/coverage-dikesfordummies-src.xml --junitxml=xunit-reports/xunit-result-
dikesfordummies-src.xml
          poetry run coverage xml -i

      - name: Autoformat code if the check fails
        if: ${{ (matrix.os == 'ubuntu-latest') && (matrix.python-version == 3.10) }}
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}  # Needed to get PR information, if any
        run: |
          poetry run isort .
          poetry run black .
          git config --global user.name '${{ github.actor }}'
          git config --global user.email '${{ github.actor }}@users.noreply.github.com'
          git remote set-url origin https://x-access-token:${{ secrets.GITHUB_TOKEN }}@github.com/$GITHUB_REPOSITORY
          git checkout $GITHUB_HEAD_REF
          git commit -am "autoformat: isort & black" && git push || true

      - name: SonarCloud Scan
        uses: SonarSource/sonarcloud-github-action@master
        if: ${{ (matrix.os == 'ubuntu-latest') && (matrix.python-version == 3.10) }}
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}  # Needed to get PR information, if any
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
```

You need to save the above content as a .yml file in the .github/workflows/ directory.

This pipeline will be executed during (any) pull-request and will ensure of several things: - The poetry installation works - The tests (all of them) are run correctly. - If the tests are all succesful: - We will verify and format all the python files in our project. - We will run a SonarCloud Scan (Follow the steps in the admin page of SonarCloud)

In addition, thanks to commitizen we can also add a step to 'bump' the package version and create new entries of the changelog. However, it is also possible as a manual step: `cz bump --changelog`.

3.5.5 Summary

Although there is still much more to see, we have seen enough resources to create our own test suite and execute it either in a GitHub workflow, or in a TeamCity step.

Now it is time to create tests and searching for more ways of providing quality to your tool.

Copyright © 2022 Carsopre

3.6 Chapter 06. Creating interfaces.

It is time to work on an interface that provides an end user access to our tool when using it from command line, or later as an .exe.

We will just focus on two options.

- CLI - like, using the click library.
- GUI - like, using the QT library.

Other option would be to make an API, but we will not cover it here.

3.6.1 Know your audience.

We have talked about this before, let's check again the potential uses of this package and its requirements:

Usage / Requirements	Endpoints	Built
Sandbox	-	-
Library	-	-
CLI	x	(Not necessarily)
API	x	(Running as a service)
GUI	x	x

As we can see, endpoints are required for anything where the user is not 'coding'. Therefore it is a good idea to create our own internal application interface so that the endpoints can easilly call the required workflows.

It is recommended to leave your `core` functionality as a separate library. This will allow you to develop separate interfaces that **uses / imports** your package. Allowing for better separation of concerns.

Of course if you just want to build a GUI or an API from the very beginning you could just go ahead and split everything in the project tree.

3.6.2 Application workflows.

As mentioned, when not working in a 'sandbox' we need to provide certain workflows (or user cases) that provide an endpoint to the user. This endpoint can be later used by an CLI, API or GUI.

We will just define in a file a workflow that given some (optional) input will display or save a profile geometry. This workflow will work as an example for building a CLI or a GUI in the following steps.

```
from pathlib import Path
from typing import List, Optional

from dikesfordummies import dike_plot
from dikesfordummies.dike.dike_input import DikeInput
from dikesfordummies.dike.dike_profile_builder import DikeProfileBuilder

def plot_dike_profile(dike_input: List[float], outfile: Optional[Path]) -> None:
    """
    Generates a `DikeProfile` plot with the reference data given in `dike_input`. The plot is either shown or saved depending on whether the argument
`outfile` is given or not.

    Args:
        dike_input (List[float]): List of values representing a Dike's profile data.
        outfile (Optional[Path]): File path where to save the plot.
    """
    _dike_input = DikeInput.from_list(dike_input)
    _dike = DikeProfileBuilder.from_input(_dike_input).build()
    _plot = dike_plot.plot_profile(_dike)
    if not outfile:
        _plot.show()
        return
    elif outfile.is_file():
```


Copyright © 2022 Carsopre

```
        outfile.unlink()
    if not outfile.parent.exists():
        outfile.parent.mkdir(parents=True)
    _plot.savefig(outfile)
```

3.6.3 CLI with Click

To simplify our project structure, we will just create a `main.py` in the *dikesfordummies* directory.

Creating an endpoint.

Click provides us with a simple way of processing the information coming from command line and connecting it to our library.

```
poetry add click
```

First, we need to make our file recognisible and 'executable':

```
if __name__ == "__main__":
    ...
```

Now the command line should be able to pass their arguments. In `Click` we do this with `@click.command` and `@click.option`:

```
from dikesfordummies import workflows

@click.command()
@click.option(
    "--dike_input",
    nargs=10,
    default=_default_input.values(),
    type=float,
    help=f"List of {len(_default_input.keys())} values for the dike input. Values represent {_dike_keys}.",
)
@click.option(
    "--outfile",
    type=click.Path(path_type=Path),
    help="The (optional) path where to save the profile plot.",
)
def plot_profile(dike_input: List[float], outfile: Optional[Path]):
    workflows.plot_dike_profile(dike_input, outfile)

if __name__ == "__main__":
    plot_profile()
```

Fortunately for us, `Click` already does the type checking for us. So we can assume that when an outfile parameter is given, then it will be of type `pathlib.Path` as specified in the argument `path_type`.

We can try to run this now:

```
python dikesfordummies\main.py plot_profile --help
```

Creating multiple entry points.

You may be already wondering how to append more commands so that not just this method can be used. Well, that's easily solved by adding a `@click.group`:

```
@click.group()
def cli():
    pass


@cli.command(name="plot_profile")
@click.option(...)
@click.option(...)
def plot_profile(dike_input: List[float], outfile: Optional[Path]):
    ...

if __name__ == "__main__":
    cli()
```

And if we try again the help command the same result should show.

We can try to run this now:

Copyright © 2022 Carsopre

```
python dikesfordummies\main.py plot_profile --help
```

Debugging from CLI.

It is relatively easy to extend the current settings to include a 'one-off' CLI call in our `launch.json`.

```
{
    "version": "0.2.0",
    "configurations": [
        ...
        {
            "name": "CLI plot default dike",
            "type": "python",
            "request": "launch",
            "console": "integratedTerminal",
            "cwd": "${workspaceFolder}",
            "program": "${workspaceFolder}\\dikesfordummies\\main.py",
            "args": [
                "plot_profile",
                "--outfile",
                "dike_plot.png"
            ],
            "justMyCode": true,
        },
        ...
    ]
}
```

Testing our CLI.

Let's not forget about tests for our endpoints:

```
def test_given_valid_input_generates_default_profile(request: pytest.FixtureRequest):
    # 1. Define test data.
    _test_dir = test_results / request.node.name
    _test_file = _test_dir / "profile.png"

    shutil.rmtree(_test_dir, ignore_errors=True)
    _args = ["--outfile", _test_file]

    # 2. Run test.
    _run_result = CliRunner().invoke(main.plot_profile, _args)

    # 3. Verify expectations.
    assert _run_result.exit_code == 0
    assert _test_file.is_file()
```

3.6.4 GUI with QT

Building a GUI in QT has its limitations if we compare to other great frameworks such as WPF. However, this should not prevent us from achieving our goals.

To simplify this project we will just create the logic in a directory **within** *dikesfordummies*. In a production project you should try to have at least a separation of concerns such as:

Because we are just *playing around* we will just create a new level in the project tree such as:

```
    \dikesfordummies
        \core
            \dike
                ...
            ...
            main.py
        \gui
            ...
            main.py
        __init__.py
    \tests
        \core
            ...
            main.py
        \gui
            ...
            main.py
        __init__.py
```


Copyright © 2022 Carsopre

Creating a basic GUI

With QT, all classical components are available. We will an interface that represents the previous endpoint.

```
poetry add pyqt5
```

We will create a new `main.py` which will contain GUI logic.

```
from PyQt5 import QtWidgets

class MainWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        super(MainWindow, self).__init__(parent)

def main():
    app = QtWidgets.QApplication(sys.argv)
    screen = MainWindow()
    screen.show()
    sys.exit(app.exec_())

if __name__ == "__main__":
    main()
```

Debugging from CLI

We can also add a debug setting:

```
{
    "version": "0.2.0",
    "configurations": [
        ...
        {
            "name": "Run GUI Main window",
            "type": "python",
            "request": "launch",
            "console": "integratedTerminal",
            "cwd": "${workspaceFolder}",
            "program": "${workspaceFolder}\\dikesfordummies\\gui\\main.py",
            "args": [],
            "justMyCode": true,
        },
        ...
    ]
)
```

Adding workflows.

Let's make it simple, we want to plot, or save the default geometry. We can demonstrate that with two simple buttons:

```
class MainWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        super(MainWindow, self).__init__(parent)
        self.setWindowTitle("Dikes For Dummies")
        self._set_menu_options()

    def _set_menu_options(self) -> None:
        self._create_menu_button(
            50,
            "Output directory",
            "Select output directory for plot(s).",
            self._get_output_file,
        )
        self._create_menu_button(
            100, "Plot", "Plot default profile", self._plot_profile
        )

    def _get_output_file(self) -> None:
        _output_dir = QtWidgets.QFileDialog.getExistingDirectory(
            self, "Select output plot directory."
        )
        if _output_dir:
            self._output_dir = Path(_output_dir)

    def _create_menu_button(
        self,
        ay_pos: float,
        title: str,
        tooltip: str,
        event: Callable,
        enabled: bool = True,
    ) -> QtWidgets.QPushButton:
        return utils.create_menu_button(
            self, dict(ax=50, ay=ay_pos, aw=160, az=30), title, tooltip, event, enabled
```


Copyright © 2022 Carsopre

```
        )

    def _plot_profile(self):
        # call DFS
        _outfile = None
        if self._output_dir:
            _outfile = self._output_dir / "default_plot.png"
        workflows.plot_dike_profile(list(workflows._default_input.values()), _outfile)
```

With this, we are good to go. We have now a very simple GUI that connects to the rest of the library.

Testing

Depending on your approach about testing gui's you may not need to do too much work. For our case, we can just simply test the initialization and plotting of a profile:

```
def test_gui(request: pytest.FixtureRequest):
    # 1. Define test data.
    _mw = MainWindow(parent=None)
    _test_dir = test_results / request.node.name
    shutil.rmtree(_test_dir, ignore_errors=True)

    # 2. Run test.
    _mw._output_dir = _test_dir
    _mw._plot_profile()

    # 3. Verify expectations
    assert _test_dir.is_dir()
    assert any(_test_dir.glob("*.png"))
```

3.6.5 Summary

That's it for this chapter. We have seen what the principles are towards creating interfaces in a python project.

Copyright © 2022 Carsopre

3.7 Chapter 07. Creating documentation

We will create now very simple documentation. What we will do is to generate technical documentation from our `docstrings`, our changelog, and some simple usage instructions

3.7.1 Mkdocs

For this chapter we will only be using Mkdocs. As described in their page, Mkdocs creates static HTML pages that can be published anywhere. This can also be integrated in a GitHub workflow

Installing the dependencies.

```
poetry add mkdocs --group dev
poetry add mkdocs-material --group dev
poetry add mkdocstrings-python --group dev
```

Defining a theme.

We will need a configuration file `mkdocs.yml` describing the way of building our documentation and its theme.

```
site_name: Dikes for dummies documentation
theme:
  name: material
  language: en
  icon:
    logo: fontawesome/solid/house-tsunami
  palette:
    - scheme: dummies
      toggle:
        icon: material/lightbulb-outline
        name: Switch to dark mode
    - scheme: slate
      toggle:
        icon: material/lightbulb
        name: Switch to light mode
  features:
    - navigation.instant
    - navigation.tracking
    - navigation.tabs
    - navigation.sections
    - navigation.expand
    - navigation.top
plugins:
  - search
  - autorefs
  - mkdocstrings:
      default_handler: python
      handlers:
        python:
          rendering:
            show_root_toc_entry: false
            show_source: true
            show_signature_annotations: true
            heading_level: 3
            show_category_heading: false
            group_by_category: false
          selection:
            inherited_members: false

      custom_templates: templates
watch:
    - dikesfordummies/
markdown_extensions:
  - pymdownx.highlight
  - pymdownx.superfences
  - admonition
  - toc:
      permalink: true
repo_url: https://github.com/Carsopre/dikes-for-dummies
repo_name: carsopre/dikes-for-dummies
extra:
  social:
    - icon: fontawesome/brands/github
      link: https://github.com/Carsopre/dikes-for-dummies
      name: Source code
copyright: Copyright &copy; 2022 Carsopre
```


Copyright © 2022 Carsopre

Building and serving

It is required to build the documentation in order to serve it

```
poetry run mkdocs build
poetry run mkdocs serve
```

Your local machine should now work as a localhost for the documentation pages.

Adding user's documentation.

In \docs we need to create a landing page named `index.md`. Within this directory we can now create as many documentation as we want either spread in files across the \docs root or in different subdirectories.

Note that if you already configured commitizen you should have also a `changelog.md` file in the docs root. Otherwise, try the following: `cz changelog`

Keep in mind only those commits following the `commitizen` rules will be shown.

Adding technical documentation.

Creating a technical reference it's easier done than said. Just mirror your solution tree-directory and create one *.md page per module. For instance, if we have:

```
\dikesfordummies
    \dike
        dike_input.py
        dike_profile.py
        dike_profile_builder.py
```

And we want to show their docstrings, we will create the following document `\docs\reference\dike.md` containing:

```
# Dike models for the Dikes for Dummies package
Technical documentation for the classes and methods within the /dike module.

## Dike Profile
::: dikesfordummies.dike.dike_profile

## Dike Profile Builder
::: dikesfordummies.dike.dike_profile_builder

## Dike Input
::: dikesfordummies.dike.dike_input
```

3.7.2 GitHub Pages

If your project is public and documentation built with MkDocs, there is no reason why you should not want to do this, as it's easy and fast.

Workflow

Create a new workflow in `.github\workflows\` with the settings required to install and run your tool

```
name: docs
on:
  push:
    branches:
      - master
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - uses: actions/setup-python@v2
        with:
          python-version: '3.10'

      - name: Run image
        uses: abatilo/actions-poetry@v2.0.0
        with:
          poetry-version: 1.2.2
      - name: Cache Poetry virtualenv
```


Copyright © 2022 Carsopre

```
        uses: actions/cache@v1
        id: cache
        with:
          path: ~/.virtualenvs
          key: venv-${{ matrix.os }}-${{ matrix.python-version }}-${{ hashFiles('**/poetry.lock') }}
          restore-keys: |
            venv-${{ matrix.os }}-${{ matrix.python-version }}-

      - name: Set Poetry config
        run: |
          poetry config virtualenvs.in-project false
          poetry config virtualenvs.path ~/.virtualenvs

      - name: Install Dependencies
        run: poetry install
        if: steps.cache.outputs.cache-hit != 'true'

      - run: poetry run mkdocs gh-deploy --force
```

Settings

We need to enable the feature in GitHub: `settings -> Pages -> Build and Deployment` - Source `deploy from a branch` - Branch `gh-pages -> / (root)` (you may require to do a first build).

If everything went well you should be able to access your built documentation in https://{your-organization}.github.io/{your-repo-name}/

Copyright © 2022 Carsopre

3.8 Chapter 08. Building the tool

We come to our last step. We already cover the different types of audiences we may have. But let's revise it from Chapter 06:

Usage / Requirements	Endpoints	Built
Sandbox	-	-
Library	-	-
CLI	x	(Not necessarily)
API	x	(Running as a service)
GUI	x	x

We only require to really build an exe when having a stand alone GUI. However we will see two ways of delivering our tool here: 1. As a built package. 2. As an exe.

3.8.1 Building and publishing our package.

Throughout the entire Dikes For Dummies workshop we have been using `Poetry`. If your dependencies are still holding up and your project is well structure you should not have too much troubles building and publishing it. Let's check it.

```
poetry check
```

All set!

```
poetry build & poetry publish
```

You will be required to authenticate yourself in pypi

Notice that most likely a `dist` directory has been created in your root directory with the wheels to be published. After publishing our package we should be able to add it as a dependency on other projects!

```
poetry add dikes-for-dummies
```

3.8.2 As an exe

This step will require (a bit) more of work. First we require the `pyinstaller` package (`poetry add pyinstaller --group dev`).

An ideal world

In theory, the following should be possible:

- Building only the CLI: `poetry run pyinstaller dikesfordummies\main.py`
- Building with GUI: `poetry run pyinstaller dikesfordummies\gui\main.py`

However, it is entirely possible that as more complex your repository starts to be, the more dependencies you need to specify by yourself. This might result on you having to create your custom `main.spec` file and your own compilation script for `pyinstaller`. We will describe these steps in the next sections. For that, lets create both files in a `\makefile` dir in our root.

init.py

Because we want our scripts to be findable and executable, we can create an `__init__.py` file that will also provide us some help:

```
from pathlib import Path

import dikesfordummies

```


Copyright © 2022 Carsopre

```
_makedir = Path(__file__).parent
_dfd_version = dikesfordummies.__version__
```

main.spec

```
# -*- mode: python -*-
from PyInstaller.utils.hooks import collect_data_files, collect_dynamic_libs
import glob, os
from pathlib import Path
from makefile import _makedir

_conda_env = os.environ['CONDA_PREFIX']

_root_dir = _makedir.parent
_dfd_src = _root_dir / "dikesfordummies"

a = Analysis([r"..\\dikesfordummies\\gui\\main.py"],
             pathex=['.', str(_dfd_src), _conda_env],
             hiddenimports=[],
             hookspath=None,
             runtime_hooks=None,
             datas=[],
             binaries= collect_dynamic_libs("rtree"),)

for d in a.datas:
    if 'pyconfig' in d[0]:
        a.datas.remove(d)
        break

print("Generate pyz and exe")
pyz = PYZ(a.pure)
exe = EXE(pyz,
          a.scripts,
          a.binaries,
          a.zipfiles,
          a.datas,
          name='DikesForDummies.exe',
          debug=False,
          strip=False,
          upx=True,
          console=False,
        )
```

Defining our custom compiler

```
import datetime
import os
import shutil
import sys
from pathlib import Path

from makefile import _dfd_version, _makedir

# __version__ is automatically updated by commitizen, do not change manually.
# Run insted 'cz bump --changelog' and then 'git push --tags' and 'git push'.
_version_file = _makedir / "version.txt"
_main_spec = _makedir / "main.spec"

def read_revision():
    # date
    now = datetime.datetime.now()
    if _version_file.is_file():
        _version_file.unlink()

    _version_as_string = _dfd_version.replace(".", ",")
    _vs_version_info = f"{_version_as_string}, 0"
    with _version_file.open("w") as f:
        f.write(
            "VSVersionInfo(\n"
            + "ffi=FixedFileInfo(\n"
            + f"filevers=({_vs_version_info}),\n"
            + f"prodvers=({_vs_version_info}),\n"
            + "mask=0x3f,\n"
            + "flags=0x0,\n"
            + "OS=0x40004,\n"
            + "fileType=0x1,\n"
            + "subtype=0x0,\n"
            + "date=(0, 0)\n"
            + "),\n"
            + "kids=[\n"
            + "StringFileInfo(\n"
            + "[\n"
            + "StringTable(\n"
            + "u'040904B0',\n"
            + "[StringStruct(u'CompanyName', u'Dummies'),\n"
            + "StringStruct(u'FileDescription', u'DikesForDummies'),\n"
            + "StringStruct(u'InternalName', u'DikesForDummies'),\n"
            + "StringStruct(u'LegalCopyright', u'Dummies"
```


Copyright © 2022 Carsopre

```
            + r" \xae "
            + str(now.year)
            + "'),\n"
            + "StringStruct(u'OriginalFilename', u'DikesForDummies.exe'),\n"
            + "StringStruct(u'ProductName', u'DikesForDummies"
            + r" \xae "
            + "Dummies'),\n"
            + f"StringStruct(u'ProductVersion', u'{_dfd_version}')])\n"
            + "]), \n"
            + "VarFileInfo([VarStruct(u'Translation', [1033, 1200])])\n"
            + "]\n"
            + ")"
        )

def compile_code():
    def _remove_if_exists(dir_name: str):
        _dir_to_remove = _makedir.parent / dir_name
        if (_dir_to_remove).is_dir():
            shutil.rmtree(_dir_to_remove)

    _remove_if_exists("dist")
    _remove_if_exists("build")
    _py_installer_exe = Path(sys.exec_prefix) / "Scripts" / "pyinstaller.exe"
    assert _py_installer_exe.is_file()
    os.system(f"{_py_installer_exe} --clean {_main_spec}")
    _version_file.unlink()

def run_compilation():
    read_revision()
    compile_code()

if __name__ == "__main__":
    run_compilation()
```

Let's run it!

```
poetry run python makefile\version_compile.py
```

After some time you should find in /dist your DikesForDummies.exe

Extending our poetry config

We can create a 'shortcut' to our compilation script so that with a single `poetry run build-exe` call everthing is executed within our safe virtual environment.

```
[tool.poetry.scripts]
build-exe = "makefile.version_compile:run_compilation"
```

3.8.3 Summary

We have finaly covered all steps required to build an MVP. There are many different ways to come to this final step, however, that is the nice challenge about python. Explore all its possibilities and don't be shy about asking or sharing your progress. Happy coding!

Copyright © 2022 Carsopre