Name of Project:	AIXS
Name of Product: (if available):	

□ Document is signed electronically on the cover page. For electronic signatures, the roles as given in the signature table below apply.

□ Document is signed manually in the signature table below.

	Role:	**Name:**	**Date:** (dd.mm.yyyy)	**Signature:**
Created:	SW Consultant SW Architect	Julian Goeser Dr.-Ing. Christoph Dinh	18.10.2022	
Reviewed:	SW Quality Manager			
Approved:	SW Program Manager			

Change History

□ The document change history is recorded in the electronic document management system (DMS) and printed on the cover page. The table below is obsolete.

□ The document is not managed in the electronic document management system (DMS). The document change history is given below.

Version Number	Changed Section	Change Description & Rationale
1.0	n.a.	Initial version

Content

1 General

1.1 Purpose

The purpose of this document is to ensure that python code is

- free of common types of errors
- free of common security issues
- readable and maintainable by different programmers
- portable and easy to adapt to changing requirements
- consistent in style

To achieve these, this document provides fixed rules that **must** be adhered to as well as recommendations and best practices that have shown to improve code quality in any of the dimensions above.

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
— Martin Fowler

1.2 Scope

This document is addressed to all software engineers involved in the production of Python code for the AIXS project.

1.3 External Documents

N.A.

1.4 Abbreviations and Acronyms

Abbreviation	Definition
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
OOP	Object-Oriented Programming
PIP	Package Installer for Python
PEP	Python Enhancement Proposals

1.5 Figures

N.A.

1.6 Applicable Standards

N.A.

1.7 Code Reviews

Quality Assurance Regulation Systems state that coding standards are mandatory for any organization developing software with quality goals. To document that our code conforms to the standard we are obliged to conduct code reviews. Besides that, code reviews are useful to detect certain kinds of programming errors which aren't likely to be found by an automated system or component tests.

2 PEP8 Compliance

PEP8 is the official python style guide. Many of the examples below were derived from it. However, some of its conventions have historical reasons, for example the 79-character line limit derived from punch cards.
The guidelines below take precedence over PEP8. For (new) language features, if they are not explicitly mentioned below, the applicable standard to follow shall be PEP8 [1].

3 Code Style

The purpose of the following style guide (section 3) is to ensure consistency in a codebase. The benefit is an increase in readability and maintainability of the produced code, especially when compared to a situation where each programmer adopts his own conventions.

3.1 Formatting

CL1 **Indentation**

Four (4) spaces per level. No tabs. Align wrapped elements vertically.

```
# Correct:

# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# Add 4 spaces (an extra level of indentation)
to distinguish arguments from the rest.
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# Hanging indents should add a level.
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

```
# Wrong:

# Arguments on first line forbidden when not
using vertical alignment.
foo = long_function_name(var_one, var_two,
    var_three, var_four)

# Further indentation required as indentation is
not distinguishable.
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

Conditionals spanning multiple lines must be distinguishable from code coming afterwards.

```
# No extra indentation.
if (this_is_one_thing and
    that_is_another_thing):
    do_something();

# Add a comment, which will provide some distinction in editors
# supporting syntax highlighting.
if (this_is_one_thing and
    that_is_another_thing):
    # Since both conditions are true, we can frobnicate.
    do_something()

# Add some extra indentation on the conditional continuation line.
if (this_is_one_thing
        and that_is_another_thing):
```

```
    do_something()
```

CL2 Tabs or Spaces?

Spaces only.

CL3 Maximum Line Length

Limit all lines to a maximum of 120 characters.
The historical line limit of 79 characters (related to max. punch card size) is not relevant in the 21. century.
Wrap longer lines with backslashes.
Example:

```
with open('/path/to/some/file/you/want/to/read') as file_1, \
     open('/path/to/some/file/being/written', 'w') as file_2:
    file_2.write(file_1.read())
```

Make sure to indent the continued line appropriately.

CL4 Should a Line Break Before or After a Binary Operator?

Break lines before the next operator.

```
# Correct:
# easy to match operators with operands
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

```
# Wrong:
# operators sit far away from their operands
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

CL5 Blank Lines

Surround top-level function and class definitions with two blank lines.
Method definitions inside a class are surrounded by a single blank line.

CL6 Source File Encoding

UTF-8. Avoid noisy unicode characters, e.g. z͡algo, and emoticons.

CL7 Imports

- Imports are at the top of the file, after possible modules comments/docstrings.
- Always use absolute imports
- Avoid wildcard imports (*)

Importing modules should usually be on separate lines:

```
import os
import sys
```

Imports from a single module can be on one line:

```
from subprocess import Popen, PIPE
```

CL8 Module Level Dunder Names

Place module level “dunders” after module docstring and below non-future imports:

```
"""This is the example module.
This module does stuff.
"""
from __future__ import barry_as_FLUFL

__all__ = ['a', 'b', 'c']
__version__ = '0.1'
__author__ = 'Cardinal Biggles'

import os
import sys
```

CL9 **String Quotes**

When a string contains single or double quote characters, use the other one to avoid backslashes in the string.

CL10 **Long files**

Do not have overly long files. As a rule of thumb, a single file should not exceed more than 2000 statements.

3.2 Naming

NC1 **Overriding Principle**

Names that are visible to the user as public parts of the API should follow conventions that reflect usage rather than implementation.

NC2 **Names to Avoid**

Avoid names that include “l” “O” or “I” or other characters that are indistinguishable from each other in certain fonts if the names become indistinguishable.

NC3 **ASCII Compatibility, Unicode**

Identifiers used must be ASCII compatible.

NC4 **Package and Module Names**

Short, all-lowercase. Underscores if they improve readability.

NC5 **Class Names & Type Variables**

Class names should normally use the CapWords/CamelCase convention.

NC6 **Type Variable Names**

Type variables should use CapWords/CamelCase.

NC7 **Exception Names**

Exceptions should be classes. CapWords/CamelCase is therefore applicable.
If the Exceptions is an error the “Error” suffix should be used.

NC8 **Global Variable Names**

Global Variables should be avoided (see the section on globals below for exceptions) at all costs.
Global Variables should be all-uppercase with underscores between words.

NC9 **Function and Variable Names**

Function and variable names should be lowercase, with words separated by underscores as necessary to improve readability.

NC10 **Function and Method Arguments**

`self` for instance methods should be the first argument. `cls` for class methods should be the first argument.

NC11 **Method Names and Instance Variables**

Use the function naming rules.
Use one leading underscore only for non-public methods and instance variables.

NC12 **Constants**

Constants are usually defined on a module level and written in all-capital letters with underscores separating words.
Examples include `MAX_OVERFLOW` and `TOTAL`.

NC13 **Designing for Inheritance**

Class attributes and methods that are for internal use only should start w. a leading underscore.
Class attributes and methods that are not intended to be passed down to a subclass should start w. two leading underscores.

3.3 Whitespace in Expressions and Statements

WH1 **Extraneous whitespace**

Avoid extraneous whitespace in the following situations.

Immediately inside parentheses, brackets or braces:

```
# Correct:
spam(ham[1], {eggs: 2})
```

```
# Wrong:
spam( ham[ 1 ], { eggs: 2 } )
```

Between a trailing comma and a following close parenthesis:

```
# Correct:
foo = (0,)
```

```
# Wrong:
bar = (0, )
```

Immediately before a comma, semicolon, or colon:

```
# Correct:
if x == 4: print(x, y); x, y = y, x
```

```
# Wrong:
if x == 4 : print(x , y) ; x , y = y , x
```

Immediately before the open parenthesis that starts an indexing or slicing:

```
# Correct:
dct['key'] = lst[index]
```

```
# Wrong:
dct ['key'] = lst [index]
```

More than one space around an assignment (or other) operator to align it with another:

```
# Correct:
x = 1
y = 2
long_variable = 3
```

```
# Wrong:
x             = 1
y             = 2
long_variable = 3
```

Avoid trailing whitespace anywhere.

WH2 Other Recommendations

Always surround these binary operators with a single space on either side:

```
# Correct:
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

```
# Wrong:
i=i+1
submitted +=1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

Function annotations should have spaces around the type hint annotations if present:

```
# Correct:
def munge(input: AnyStr): ...
def munge() -> PosInt: ...
```

```
# Wrong:
def munge(input:AnyStr): ...
def munge()->PosInt: ...
```

Don’t use spaces around the = sign when used to indicate a keyword argument, or when used to indicate a default value for an *unannotated* function parameter:

```
# Correct:
def complex(real, imag=0.0):
    return magic(r=real, i=imag)
```

```
# Wrong:
def complex(real, imag = 0.0):
    return magic(r = real, i = imag)
```

When combining an argument annotation with a default value, however, do use spaces around the = sign:

```
# Correct:
def munge(sep: AnyStr = None): ...
def munge(input: AnyStr, sep: AnyStr = None,
limit=1000): ...
```

```
# Wrong:
def munge(input: AnyStr=None): ...
def munge(input: AnyStr, limit = 1000): ...
```

Avoid long single line statements if they hinder read- and understandability.

```
# Acceptable in some cases:
if foo == 'blah': do_blah_thing()
for x in lst: total += x
while t < 10: t = delay()
```

```
# Wrong:
if foo == 'blah': do_blah_thing()
else: do_non_blah_thing()

try: something()
finally: cleanup()

do_one(); do_two(); do_three(long, argument,
                             list, like, this)
```

```
if foo == 'blah': one(); two(); three()
```

3.4 Enforcement

The recommended, and easiest, way to enforce the rules defined above is to adopt a compliant formatter. The tooling section below has some recommendations.

4 Readability and Understanding

Increasing the readability and understandability of source code improves the speed at which developers debug issues, write new features and work with code in general. Therefore, optimizing these directly affects the performance of the team.

4.1 Useful Names

This section contains a set of conventions on how to choose, write and administer names for all entities over which the programmer has control.

> *"There are 2 hard problems in computer science: cache invalidation, naming things, and off-by-1 errors."*
> *- Martin Fowler*

NM1 **Pronounceable Names**

Prefer pronounceable names over abbreviations. They are easier to read and easier to talk about. Try to give voice to Example 2 for a comparison.

Example 1: Use name_length instead of nln.
Example 2: Use none_random_int instead of n_rnd_intgr.

Exception: If the meaning of the identifier is well-known (e.g., "min" for "minimum", "i" for "index"), and the identifier is limited in scope (e.g., local to a function), use of an abbreviation or acronym is acceptable.

NM2 **Descriptive Names**

Use names that are English and self-descriptive.
Don't use terms from other languages.
Don't use overly long or complicated class names, even if they would be self-descriptive.
A class named *DonauDampfschiffahrtGesellschaftsKapitän* is not very readable.

NM3 **Meaningful Names**

Names of classes, methods and variables must be meaningful and must reflect their intended use.

Use **nouns** for class names.
Example: `PatientRecord` or `GraphicsManager`.
Use **verbs** for methods, functions.
Example: `open_plan_file; show_statistics.`

The "-ed" ending for boolean typed variables can be useful such as initialized or calculated.

NM4 **Similar Names**

Do not use very similar names. Very similar names might cause confusion in reading the code.
Do not create names that differ only by case.

Example: Avoid this: cm_lower, cs_lower.
track, Track, TRACK

> *"You should name a variable using the same care with which you name a first-born child."*
> *- Robert C. Martin*

4.2 Comments

Good comments ease working with pieces of code that a developer has not written himself. They clarify design decisions, can be used for automatic documentation and allow an instant glimpse of the commented codes purpose.

Bad, or too many comments can clutter the code, hinder readability or even worse, provide misleading information. It is therefore beneficial to understand when and how to write comments for a specific audience and purpose.

> *"Every time you write a comment, you should grimace and feel the failure of your ability of expression."*
> *- Robert C. Martin*

NC1 Types of Comments

Comments in source serve a different purpose depending on where they are placed.

Docstrings

Located in functions, modules or packages. They explain what the function/package/module does beyond what can be discovered by introspection. All parameters, return values and exceptions are clearly described.
Docstrings should also mention any side-effects that could happen as a result of calling the function.
They are **for the users** of your code.

Inline Comments

Inline comments explain why code was written the way it is written.
Code that is self-explanatory does not require inline comments.
They are **for the developers** maintaining or reading your code.

NC2 Docstring Formats

As a team, agree on a standard docstring style.
This enforces consistency and is easier for automatic documentation tools.

Example (*Googles Docstring Format*):

```
def some_function(param1: str, param2: int) -> str:
    """_summary_

    Args:
        Param1 (str): _description_
        Param2 (int): _description_

    Returns:
        str: _description_
    """

 # Implementation here
```

NC3 Useful comments

Docstrings should explain briefly what the code does. What it requires (parameters), what it returns and what exceptions are raised by it. Additionally, any side-effects that can occur should be mentioned.
As there are tools that create documentation from docstrings a good way of verifying a docstring comment is to imagine it being part of the overall system documentation.

Inline comments serve the purpose of helping understand complex pieces of code or design decision taken in the implementation. Code that is self-explanatory does not require such comments.

Examples:

```
# Bad
word = "hello" # Assign "hello"
# Very Bad
word = "Python" # A variable word that has been assigned a string value of “Python”
```

```
# Almost Ok, describes what the function does only
def is_prime(x):
    """
    is_prime(x) -> true/false.  Determines whether x is prime,
    and return true or false.
    """
# Better, adds information that is not clearly visible when reading the functions code
def is_prime(x):
    """
    Returns true if x is prime, false otherwise.

    is_prime() uses the Bernoulli-Schmidt formalism for figuring out
    if x is prime. Because the BS form is stochastic and hysteretic,
    multiple calls to this function will be increasingly accurate.
    """
```

5 Pythonic / Idiomatic Code

Idiomatic Python is written when the programmer struggles with solving the problem only, not with the programming language itself. The idioms he chooses are those that provide the obvious solution to his problems, often, but not always, there is only one obvious way to solve a task. As *obvious* lies in the eye of the beholder and the amount of valid solutions is often greater than one, the points below should be read as a guideline of good practices, not as a lawbook.

“Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”
– Brian W. Kernighan.

5.1 Structuring Code

“Walking on water and developing software from a specification are easy if both are frozen.”
– Edward V. Berard.

SC1	**Absolute Paths** Don’t use absolute or os specific file paths. They won't work on other people's machines 99% of the time.
SC2	**Virtual Environments** Always use a virtual environment for your local development. Choose the appropriate tool to manage virtual environments as a team. Combined w. proper dependency management they ensure development builds are reproducible on various operating systems and machines.
SC3	**Dependency Management, Versions** As a team, agree on how each developer manages the dependencies for his project and what tools are used to that extent. Agree on specific version of packages if necessary. Especially, agree on which python interpreter version the team uses. Ensure that builds are stable over time and reproducible.
SC4	**Packages** As a team, decide on a common folder structure/hierarchy that is used for packages of a common category. Meaning instances of services, api modules, internal libraries etc. share a common folder structure. Also decide on a naming convention for these. Add, and maintain, a README file.
SC5	**Packaging and Deploying** If the package you write is intended to be installed by others, consider writing a *setup.py* file so people can install it in a standardized way. The package will be copied into Pythons site-packages directory. As a team, agree on a standard way to package and deploy installable packages.

5.2 Declaration, Arguments, Tuples

CD1 **Single Line Declaration**

Declaring multiple variables on the same line is not recommended.
The code will be more difficult to read and understand.

CD2 **Immediate Initialization**

Declare each variable with the smallest possible scope and initialize it immediately.
It is best to declare variables close to where they are used.

CD3 **Globals**

Avoid them. Exceptions:

```
# Constants
MAX_TIMEOUT = 50
# Module level dunder constants
__author__ = "Some One <some.one@someemailprovider.com>"
# Import time computation
COPY_BUFSIZE = 1024 * 1024 if _WINDOWS else 16 * 1024
```

CD4 **Positional vs Keyword arguments**

Use Keyword arguments where they aid in understanding the function code.

```
# Bad, it is not clear what the arguments mean
twitter_search('spaetzle', 20, True, False)

# Good
twitter_search('spaetzle', numtweets=20, retweets=True, unicode=False)
```

CD5 **Named Tuples**

Prefer named tuples if they help understanding code.

```
# It is clear what the code does, it is not clear what it means
p = (170, 0.1, 0.6)
if p[1] >= 0.5:
    print 'very bright!'
if p[2] >= 0.5:
    print 'very light!'

# Using named tuples it is
from collections import namedtuple
Color = namedtuple('Color', ['hue', 'saturation', 'luminiosity'])
p = Color(170, 0.1, 0.6)

if p.saturation >= 0.5:
    print 'very bright!'
if p.luminiosity >= 0.5:
    print 'very light!'
```

5.3 Type Hints

TH1 **When to use them**

Use type hints for all function arguments and return values.
Use type hints in docstrings for parameters, arguments, exceptions and return values.
Use type hints for local variables if they don’t hinder readability.

TH2 **Purpose of Type Hints**

Understand that type hints are an annotation feature only.
The caller of type annotated function can still pass whatever type he likes to the function.

```
def sum(a: int, b: int) -> int:
    # Use type assertions to ensure a specific type is passed
    # assert isinstance(b, int), “need and integer”

    return a + b

# Works, leads to runtime exception
sum(1,’2’)
```

Another approach to enforce strong typing is to use *mypy* to verify types at import time.

5.4 Errors, Exceptions and Logging

EXC1 **Gotta Catch ‘Em all!**

Generally, catch all errors thrown by a function. Catch only those that can possibly be thrown by the code in the try block.
If it is beneficial to not catch an exception, do so explicitly.
It is usually not useful to catch exceptions at the same level in the hierarchy where they are thrown since the context of how to deal with that exceptional situation can often be known at much higher levels only.

EXC2 **Catch Specific Exceptions**

Prefer specific exceptions instead of using **BaseException** or **Exception**, avoid a bare except clause.
These will catch exceptions such as **SystemExit** and **KeyboardInterruption**.
Which makes it harder to interrupt and debug programs.

You can use general exceptions:

- If the exception handler will be printing out or logging the traceback; at least the user will be aware that an error has occurred.
- If the code needs to do some cleanup work, then let the exception propagate upwards with raise. try...finally can be a better way to handle this case.

Derive exceptions from Exception rather than BaseException. Direct inheritance from BaseException is reserved for exceptions where catching them is almost always the wrong thing to do.

EXC3 **Limit try blocks**

Limit the code in try blocks to the absolute minimum to avoid masking bugs.

```
# Correct:
try:
    value = collection[key]
except KeyError:
    return key_not_found(key)
else:
    return handle_value(value)
```

```
# Wrong:
try:
    # Too broad!
    return handle_value(collection[key])
except KeyError:
    # Will also catch KeyError raised by handle_value()
    return key_not_found(key)
```

EXC4 **Exception Chaining**

`raise X from X` should be used to indicate explicit replacement without losing the original traceback.

When deliberately replacing an inner exception (using raise X from None), ensure that relevant details are transferred to the new exception (such as preserving the attribute name when converting KeyError to AttributeError).

EXC5 **Exception Path Awareness**

Ensure that all execution paths are considered when handling exceptions.
Example:
If the except block does not return, declarations in the try block can be undefined after the except... clause.

```
try:
    Y = 100 / X
    Z = 23 * Y
except ZeroDivisionError as err:
    print(err)

# Z is undeclared if an exception is raised
return Z
```

EXC6 **Defining Custom Exceptions**

Define and agree as a team on a common exception schema that is used uniformly across all components.

EXC7 **Logging**

- Write meaningful log messages. Understand that the surrounding context (code) is missing when logs are read.
- Include a timestamp. Knowing something happened is only half as useful as additionally knowing when it happened.
- Adopt a specific format for log messages. E.g. ISO-8601. This makes logs more readable and easier to parse by additional libraries/tools.
- **Don't log/print sensitive information.**
- Be aware of the purpose of your logs. Is your log intended for debugging purposes, for performance testing, for retaining metrics etc.

5.5 Object Oriented Programming

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

– C. A. R Hoare

OO1 **Getters and Setters**

Don't write them. They come from languages that don't make use of descriptors. Python does.

If you find a sensible use case, use the built-in decorators for setters and getters.

OO2 **Private methods/attributes**

Python does not include the concept of private methods or attributes. Understand that, in most cases, people will have access to your attributes and methods.
Use leading underscores to indicate a method that should be considered private, this will make us of *name mangling:* Any identifier of the form __xxx (at least two leading underscores, at most one trailing underscore) is textually replaced with _classname__xxx, where classname is the current class name with leading underscore(s) stripped.

```
class A:
    def __private():
        print 'hello'

A._A__private()
```

OO3 **Polymorphism**

Understand that Python is implicitly polymorphic. The built-in *len* function is a good example.
Don't implement these methods if there is a built-in method.

OO4 **Inheritance**

Avoid complex inheritance hierarchies. Be aware of the diamond problem.

OO5 **Metaclasses**

Classes in python are objects. Classes can create objects.
Metaclasses are the classes that create class objects.
If you need to inject logic at the creation of a class, use metaclasses.
But make sure you really understand what you are doing.
Example: Keep track of the order in which classes are defined so that they can be instantiated in the same order later.

```
class MyMeta(type):
    counter = 0

    def __init__(cls, name, bases, dic):
        type.__init__(cls, name, bases, dic)
        cls._order = MyMeta.counter
        MyMeta.counter += 1

class MyType(metaclass=MyMeta):
    pass
```

5.6 Testing

TE1 **Terminology**

Distinguish between *Unit*, *Component*, *Integration*, *Regression* and *End2End* tests.
Separate different types of tests by moving them to different files/folders.

TE2 **Coverage**

Code coverage reports tell you the percentage of code lines executed during your tests.
Just because a test executed a line of code does not mean it has correctly asserted all behavior.
Treat code coverage therefore as a metric only that serves as indication of where tests are needed.

“If you make a certain level of coverage a target, people will try to attain it. The trouble is that high coverage numbers are too easy to reach with low quality testing.”

– C. A. R Hoare

TE3 **Test Structure**

Put tests in an extra directory outside the actual application code (for easier imports).
Add clear separation between different types of tests.
Use the features of your testing library. Example (pytest):

```
@pytest.mark.integtest
def your_test():
    pass
```

Consider grouping tests in classes if the number of tests increases.
Contrary to naming conventions above, use long descriptive names for testing functions.
Testing functions are never called explicitly, you'll never write out the name of a test function.

TE4 **Unit Tests**

Should focus on one tiny bit of functionality, be fast and isolated.
Unit tests don’t interact with other components. If the integration is intrinsically part of the function mock it.

TE5 **Factories**

Use factory classes for reusable creation of dummy input/output:

```
class HTTPRequestFactory:
    @staticmethod
    def create_post_request(
        headers: dict,
        data_payload_json: str = InputPayloadFactory.create_valid_json_str()
    ) -> func.HttpRequest:
        pass

    @staticmethod
    def create_options_request() -> func.HttpRequest:
        pass

    @staticmethod
    def create_post_request_cloudevent(cloud_event: CloudEvent) -> func.HttpRequest:
        pass

    @staticmethod
    def create_post_request_no_cloudevent(json_dictionary: dict) -> func.HttpRequest:
        pass
```

TE6 **Happiness**

Don’t forget to test the happy path.

TE7 **Fixtures and Parametrization**

While regular functions will work to implement test setup/teardown you should prefer fixtures.
Fixtures are reusable and come with more functionality for various scenarios than regular functions.
Always prefer parametrizing tests to test multiple inputs to a function and verify the expected output.
Rule of thumb: Parametrize the same behavior, different behaviors need different tests

TE8 **Cartesian Products and Permutations**

Helpful for generating all possible test parameter combinations for a function.

```
names = ["file1.txt", "file2.txt"]
modes = ["r", "w", "a"]
encodings = ["utf-8", None]

# All possible combinations
list(product(names, modes, encodings)))
```

Permutations are useful for exploring all possible orderings of elements.

```
list(permutations(['eat', 'drink', 'sleep']))
```

5.7 Performance

"Premature optimization is the root of all evil"
— Donald Knuth & Sir Tony Hoare

PE1 **List Comprehension**

Are, in general, a bit faster than operating on a list in a loop.

PE2 **Generators**

For tasks that require a lot of precomputed data, consider writing your own generator.

A generator will compute elements on demand.

PE3 **Built-ins**

Built-in functions of the standard library are often better tested and faster than the functions you write on your own. Prefer these if possible.

PE4 **File & DB Access**

Reduce the number of queries/operations on files/database. They usually are the bottleneck. Use aggregation techniques, e.g. batches, to bundle requests.

5.8 Miscellaneous

MC1 **Comparison Recommendations**

Use is not operator rather than not ... is.

```
# Readable
if foo is not None:
```

```
# Less readable
if not foo is None:
```

Don’t compare boolean values to True or False using ==:

```
# Correct:
if greeting:
```

```
# Wrong:
if greeting == True:
# Worse:
if greeting is True:
```

Use the ternary operator for simple switches.

```
# Ok:
```

```
# Better:
```

```
if weight > 100:
    category = 'overweight'
else:
    category = 'normal'
```

```
category = 'overweight' if weight > 100 else category = 'normal'
```

MC2 Mutable Default Arguments

Python handles default arguments in function definitions a bit different than expected. Default arguments are evaluated once at definition, not each time the function is called.

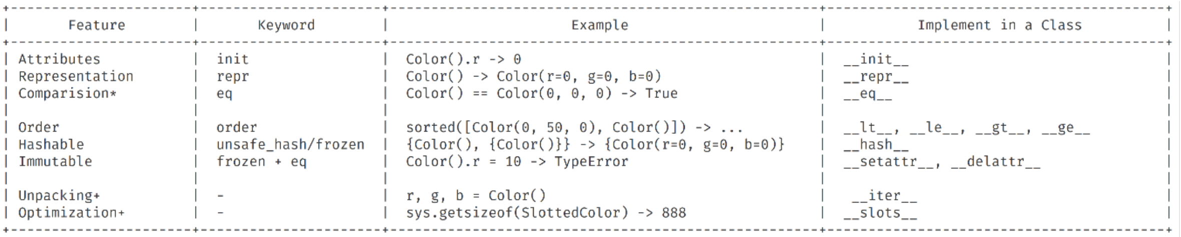
Example:

```
def append_to(element, to=[]):
    to.append(element)
    return to

my_list = append_to(12)
print(my_list)

my_other_list = append_to(42)
print(my_other_list)
```

The expected result/output is, for most, [12][42]. However, it is [12] [12, 42].
What happens is that a new list is created **once,** when the function is defined. It is then reused in each call.

MC3 Dataclasses

The default use case of dataclasses is to avoid boilerplate code. Dataclasses will generate __init__, __eq__, __hash__ and __repr__ methods automatically. Every time you create a class that mostly consists of attributes, choose dataclasses.
For more advanced use cases of dataclasses, see [2].



Feature	Keyword	Example	Implement in a Class
Attributes	init	Color().r -> 0	__init__
Representation	repr	Color() -> Color(r=0, g=0, b=0)	__repr__
Comparision*	eq	Color() == Color(0, 0, 0) -> True	__eq__
Order	order	sorted([Color(0, 50, 0), Color()]) -> ...	__lt__, __le__, __gt__, __ge__
Hashable	unsafe_hash/frozen	{Color(), {Color()}} -> {Color(r=0, g=0, b=0)}	__hash__
Immutable	frozen + eq	Color().r = 10 -> TypeError	__setattr__, __delattr__
Unpacking+	-	r, g, b = Color()	__iter__
Optimization+	-	sys.getsizeof(SlottedColor) -> 888	__slots__

MC4 Traditional Looping

Pythons loop are different than traditional loops used for example in C or Java.
Be aware of the pythonic way of looping.

Example: Looping over a range of numbers

```
# Basic
for i in [0,1,2,3,4,5]:
    print i**2

# Using range, all numbers are generated at once
for i in range(6):
    print i**2

# Using xrange, numbers are generated on demand
# Note: In python3 range and xrange are the same
for i in xrange(6):
    print i**2
```

Example: Looping over a collection

```
colors = [‘red’,’green’,’blue’]
```

```
# Dont do this, C-style looping
for i in range(len(colors)):
    print(colors[i])

# Do instead
for color in colors:
    print(color)
```

Example: Looping over multiple collections

```
colors = [‘red’,’green’,’blue’]:
names = [‘juan’,’schorsch’,’anna’,’xi’]:

# Dont do this, C-style looping
n = min(len(names), len(colors))
for i in range(n):
    print names[i], colors[i]

# Do instead
for name,color in zip(names,colors):
    print names[i], colors[i]
```

MC5 Dunder / Magic methods

Be aware of their purpose. If you create a class and need a certain functionality, check if there is a dunder method for it. Example, x and y are instances of a class implemented by you.

```
# If objects should be addable, printable etc.
# x + y   ->     use __add__
# repr(x) ->     use __repr__
# len(x)  ->     use __len__
```

MC6 When to Use Trailing Commas

Mandatory when making a tuple of one element.

```
# Correct:
FILES = ('setup.cfg',)
```

```
# Wrong:
FILES = 'setup.cfg',
```

MC7 Context Managers

Context managers should be invoked through separate functions or methods whenever they do something other than acquire and release resources:

```
# Correct:
with conn.begin_transaction():
    do_stuff_in_transaction(conn)
```

```
# Wrong:
with conn:
    do_stuff_in_transaction(conn)
```

The latter example doesn’t provide any information to indicate that the __enter__ and __exit__ methods are doing something other than closing the connection after a transaction. Being explicit is important in this case.

MC8 List Comprehension

Use list comprehension to simplify data collection tasks.

```
# Using a loop
squares = []
for x in range(10):
    s = x*x
    squares.append(s)

# Using List Comprehension
squares = [x*x for x in range(10)]
```

If the data is consumed directly use a more efficient generator expression:

```
sum(x*x for x in range(10))
```

Avoid nested list comprehensions.

MC9 Return statements

Be consistent. Either all return statements in a function should return an expression, or none of them should.

If any return statement returns an expression, any return statements where no value is returned should explicitly state this as return None, and an explicit return statement should be present at the end of the function (if reachable):

```
# Correct:

def foo(x):
    if x >= 0:
        return math.sqrt(x)
    else:
        return None

def bar(x):
    if x < 0:
        return None
    return math.sqrt(x)
```

```
# Wrong:

def foo(x):
    if x >= 0:
        return math.sqrt(x)

def bar(x):
    if x < 0:
        return
    return math.sqrt(x)
```

MC10 Expression Chains

Chains of **and** and **or** can be expressed using **any()** and **all()**

```
a and b and c and d
# becomes
all([a,b,c,d])

a or b or c or d
# becomes
any([a,b,c,d])
```

MC11 OS Dependency

Always make sure that all code, configuration, tests and tools are os agnostic.

MC12 Dependency Injection

The main idea of dependency injection is to separate the creation and usage of objects.

Example:

```
# Bad, the class is directly dependent on the db_sdk provided by create_db_sdk()
class some_service():
    db_sdk: create_db_sdk()

    def do_something(self):
        self.db_sdk.execute_something()

# Good, the creation of an sdk is moved outside the class
class some_service():
    db_sdk: SdkType
    def __init__(self, sdk_client: SdkType):
        self.db_sdk = sdk_client

    def do_something(self):
        self.db_sdk.execute_something()
```

6 Development Team Space

6.1 Tooling

There exist far more tools than are, and could be, listed here. The point of this section is to have a starting point and see what type of additional categories of tools exist that can improve some metric of your work.
There would be little benefit to setting or arguing for a specific linter in this section. The points below ensure that you are aware of the existence of the tools, and their possible benefits in general, at all.

TO1 **IDEs**

It is each individual developer's choice and responsibility to select an appropriate editor/IDE.

Popular examples include PyCharm, VSCode or Vim/NVim/Emacs.

Ensure that the codebase is not dependent on a specific IDE/editor.

TO2 **Version Control**

Use git. Developers should be familiar with its basic operation.

TO3 **Formatting Code**

Decide on which formatter to adapt as a team. Every developer should integrate it into his workflow/setup.

Use a common configuration if the default is not good enough.

Some PEP8 compliant formatters are: Black, Autopep8, yapf

TO4 **Analyzing Code**

Static-code analysis can provide many valuable insights into different aspects of your code.
Linters: pylint, Flake8
Static Type Checking: mypy
Security Analysis: bandit, pip-audit

TO5 **Testing**

Apart from the built-in *unittest* package several libraries exist for more advanced testing. For example

PyTest, Robot (Acceptance Tests), Behave (BDD), Hypothesis or Atheris

TO6 **Documentation**

The prominent tool to generate documentation from dosctrings, among other things, is Sphinx.

For APIs use Swagger or AsyncStudio.

6.2 Habits / Practices

HP1 **Version Control**

Commit frequently. Keep commits small.
Frequently merge dev/master into your feature branch.
Run your tests before opening a pull requests.
Run the git garbage collector every few months.

Dont commit bytecode (.pyc) or *__pycache__* folders

Maintain a readable *.gitignore* file. If you notice there are files beeing commited that should not be in the repository update it.

HP2 **Commit Messages and Branches**

The notions of meaningful comments and names described in the chapters above also apply to commit messages and branches. Try to describe **what** the commit changes and **how** it does it as **briefly** as possible.

```
# Don’t do this (please!)
git commit -m "Make run again."
# Don’t do this (please!)
git branch my_branch_v1.3
# Or this
git commit -m "updated"

# Better
git commit -m "Add requests dependency to fix api get test case"
# Better (Depends on branch naming convetion)
git branch 345345_hdp_integration_module
```

HP3 Cleanup

Regularly delete unused/merged branches, tempory/scratch files, imports and cloud resources.
This makes it easier for people to look for branches/files/resources and also saves costs.

HP4 Ready for work!

Ensure that your local development environment is fit for purpose!
You should be able to run, if applicable, code on your machine and have the necessary tools ready that aid in the development process.
If that is not case you will not be able to code and debug as effectively.

1. Figure out what the issue is or why it occurs on your machine
2. Raise the issue with the relevant person

HP5 Code Reviews

“Optimism is an occupational hazard of programming: feedback is the treatment.“
— Kent Beck

The purpose of code reviews is to ensure the quality of a codebase over time. They also serve as a method of training and knowledge sharing between developers.

For the *Reviewer*

> You are are responsible that the PR does not degrade code quality.
> If you make it very difficult to merge, people will be demotivated.
> If you make it too easy, quality may degrade.
>
> It is fine to add comments on style or alternative solutions, a PR does not have to be perfect in every metric. Dont delay its merge because of it.
>
> Review all aspects of the code: *Design, Functionality, Complexity, Security, Tests, Naming, Comments, Style and Documentation*.
>
> Be kind and respectful. Explain your reasoning, give guidance and accept explanations.

For the *Reviewee*

> Keep the size of your PR reasonable, 100-300 LoC are good rule of thumb.
> If you never submit an improvement the review will stagnate.
> Make sure you have taken adequate measures to prevent obvious flaws in your code (merge messages, unformatted files, etc.). You are requesting another developers valuable time.
>
> Be kind and respectful. Explain your reasoning and accept explanations.

For more suggestions on the topic see googles *code review developer guide* [3].

7 The zen of python and other resources

```
>>> 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!
```

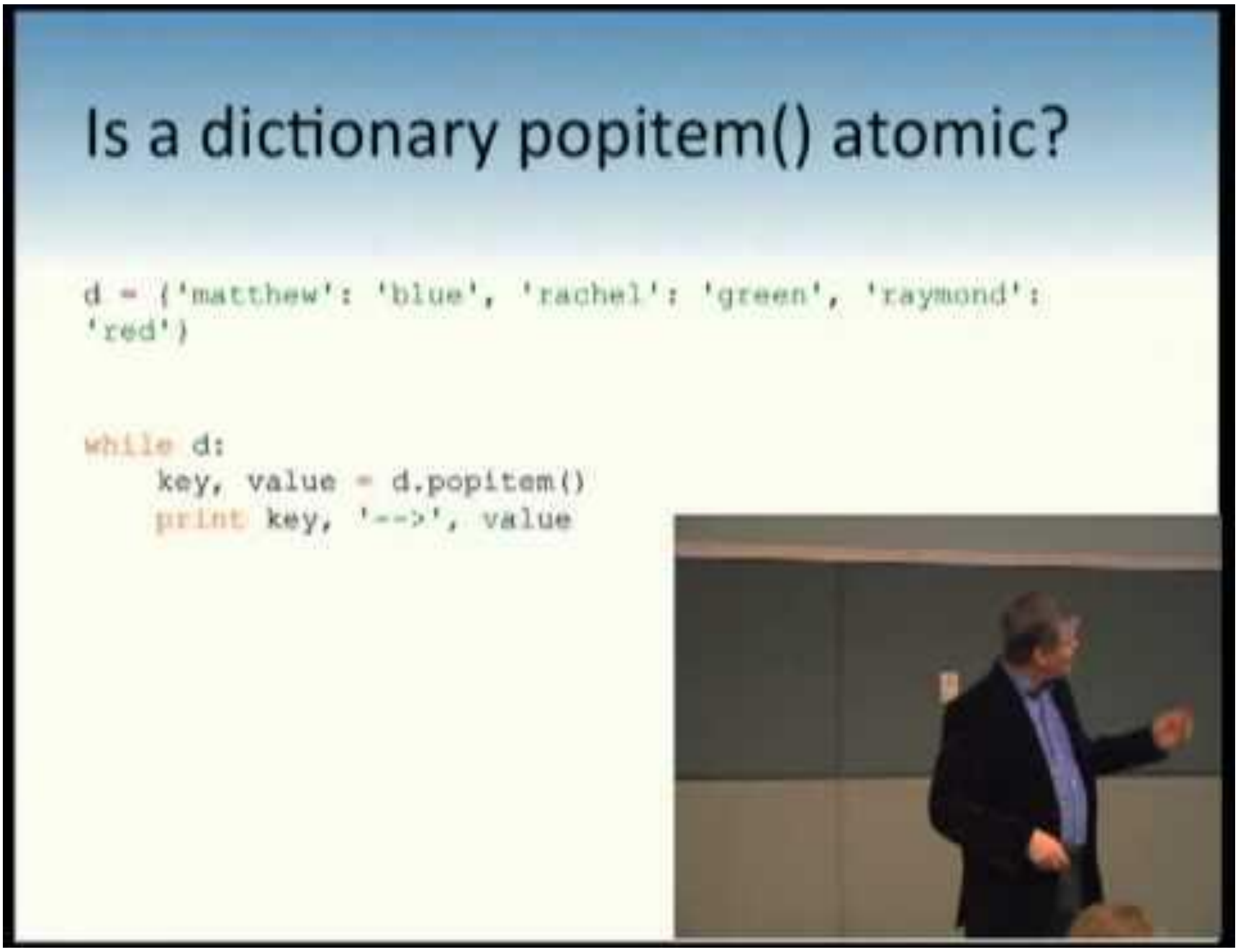
7.1 Articles

https://python-patterns.guide

7.2 Talks

https://www.youtube.com/watch?v=wf-BqAjZb8M

Transforming Code into Beautiful, Idiomatic Python



Is a dictionary popitem() atomic?

```
d = {'matthew': 'blue', 'rachel': 'green', 'raymond':
'red'}

while d:
    key, value = d.popitem()
    print key, '-->', value
```

James Powell: So you want to be a Python expert? | PyData Seattle 2017

```
class Polynomial:
    def __init__(self, *coeffs):
        self.coeffs = coeffs

    def __repr__(self):
        return 'Polynomial(*{!r})'.format(self.coeffs)

    def __add__(self, other):
        return Polynomial(*(x + y for x, y in zip(self.coeffs, other.coeffs)))

p1 = Polynomial(1, 2, 3) # x^2 + 2x + 3
p2 = Polynomial(3, 4, 3) # 3x^2 + 4x + 3
```

```
~/python-expert/data-model/data-model.py
"data-model.py" 12L, 393C [w]

  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'Polynomial' and 'Polynomial'
```

8 References

Python's development is conducted largely through the Python Enhancement Proposal (PEP) process, the primary mechanism for proposing major new features, collecting community input on issues, and documenting Python design decisions.

Python code style is covered in PEP 8 [1] of which many of this document's items were adopted from.

[1] Python Enhancement Proposals: PEP8 https://peps.python.org/pep-0008/
[2] On Dataclasses: https://stackoverflow.com/a/52283085
[3] Google Engineering Practices Documentation: https://google.github.io/eng-practices/