Here’s a concise yet detailed **PEP 8 Cheatsheet** formatted to fit on **1–2 A4 sheets**. It covers the most critical guidelines for writing clean, readable Python code:

PEP 8 Cheatsheet

Python Style Guide Essentials

---

1. Indentation & Line Length

- **Indent with 4 spaces** (never tabs).
- **Max line length**:
  - **79 characters** for code.
  - **72 characters** for docstrings/comments.
- Break long lines with `()` or `\`, or use implied line continuation (e.g., inside `[]`, `{}`, `()`).

2. Naming Conventions

- **Variables/Functions**: `snake_case` (e.g., `calculate_total`).
- **Classes**: `CamelCase` (e.g., `ClassName`).
- **Constants**: `UPPER_CASE` (e.g., `MAX_LIMIT`).
- **Private Variables**: Prefix with `_` (e.g., `_internal_value`).

3. Whitespace

- **Operators**:

```
# Good
x = 5 + 3
# Bad
x=5+3
```

- **Commas**:

```
# Good
list = [1, 2, 3]
# Bad
list = [1 , 2 ,3]
```

- **Avoid extra spaces**:

```
# Good
function(arg1, arg2)
# Bad
function( arg1, arg2 )
```

- **Blank Lines**:
  - **Two blank lines** between top-level functions/classes.
  - **One blank line** between methods in a class.

4. Comments

- **Block comments**: Start with `#` and a space.

- **Inline comments**: Use sparingly, and separate with 2 spaces:

```
x = 5  # This is an inline comment
```

- **Docstrings**:
  - Use `"""triple double quotes"""` for modules, functions, classes.
  - Write one-line docstrings on the same line.
  - For multi-line docstrings, put the closing `"""` on a new line.

5. Imports

- **Group imports** in this order:
  1. **Standard library** (e.g., `import os`).
  2. **Third-party** (e.g., `import numpy`).
  3. **Local** (e.g., `from my_module import func`).
- **Avoid wildcard imports**:

```
# Bad
from module import *
```

- **Use separate lines**:

```
# Good
import os
import sys
# Bad
import os, sys
```

6. Code Structure

- **Avoid trailing commas** unless necessary (e.g., for Git diffs).
- **Comparisons**:

```
# Use 'is' for None, True/False
if x is None:
# Use '==' for values
if x == 5:
```

- **Quotes**: Prefer `'single quotes'` unless the string contains `'`. Use `"""` for docstrings.

7. Function/Method Design

- **Limit arguments**: Use `*args` or `**kwargs` for flexibility.
- **Type hints** (optional but encouraged):

```
def greet(name: str) -> str:
    return f"Hello, {name}"
```

8. Error Handling

- **Be specific**: Avoid bare `except:` clauses.

```
# Good
except ValueError:
```

```
# Bad
except:
```

Tools for PEP 8 Compliance

- **Linters**: `flake8`, `pylint` (check code style).
- **Autoformatters**: `black` (reformat code), `autopep8` (fix PEP 8 issues).
- **IDE Plugins**: Enable PEP 8 checks in VS Code, PyCharm, etc.

Key Exceptions

- **Readability > Rules**: Break PEP 8 if it improves clarity (but document why).
- **Legacy Code**: Follow existing conventions in older projects.

Here's your expanded **PEP 8 Cheatsheet** with additional examples and a new section for indentation alignment:

PEP 8 Cheatsheet

Python Style Guide Essentials

---

1. Indentation Alignment

- **Align with opening delimiter** for function calls/definitions:

```
 # Correct: Aligned with opening delimiter
result = long_function_name(var_one, var_two,
                            var_three, var_four)

# Correct: Extra indentation for function arguments
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# Correct: Hanging indent (add one level)
result = long_function_name(
    var_one, var_two,
    var_three, var_four
)
```

---

3. Whitespace

- **Blank Lines**:

```
 # Two blank lines between top-level functions/classes
def function_one():
    pass


def function_two():
    pass
```

```python
# One blank line between methods in a class
class MyClass:
    def method_one(self):
        pass

    def method_two(self):
        pass
```

---

6. Avoid Trailing Commas

- **Use trailing commas only for multi-line lists/tuples** (to simplify Git diffs):

```python
# Correct: No trailing comma for single-line
colors = ['red', 'green', 'blue']

# Correct: Trailing comma for multi-line (optional but helpful)
colors = [
    'red',
    'green',
    'blue',  # Trailing comma makes adding new items easier
]
```

---

7. Function/Method Design

- **Limit arguments** using `*args` or `**kwargs`:

```python
# Avoid too many arguments
def bad_design(name, age, address, phone, email):
    pass

# Better: Use *args or **kwargs for flexibility
def better_design(name, age, **additional_info):
    print(f"{name}, {age}")
    print(additional_info)

# Example usage
better_design("Alice", 30, city="Paris", country="France")
```

Other Key Conventions

- **Type hints** (optional but encouraged for clarity):

```python
def calculate_total(items: list[int], discount: float = 0.1) -> float:
    return sum(items) * (1 - discount)
```

Tools for PEP 8 Compliance

- **Linters**: `flake8`, `pylint`
- **Autoformatters**: `black`, `autopep8`
- **IDE Plugins**: Enable in VS Code, PyCharm, etc.

When to Break PEP 8

- **Readability first**: Break rules if it makes code clearer (e.g., aligning columns in data).
- **Legacy code**: Follow existing conventions in older projects.

Type hints (also known as **type annotations**) are a feature in Python that allow you to explicitly specify the expected data types of variables, function arguments, and return values. They were introduced in **Python 3.5** (PEP 484) and are optional but highly recommended for improving code clarity, readability, and maintainability.

Type hints do not enforce type checking at runtime (Python remains dynamically typed), but they help tools like linters, IDEs, and static type checkers (e.g., `mypy`) catch potential type-related errors before running the code.

Why Use Type Hints?

1. **Improved Readability**: Type hints make it clear what types of data a function expects and returns.
2. **Better Tooling**: IDEs and linters can provide better autocompletion, error detection, and refactoring support.
3. **Early Error Detection**: Static type checkers like `mypy` can catch type mismatches before runtime.
4. **Documentation**: Type hints serve as a form of documentation, making it easier for others (and your future self) to understand the code.

Basic Syntax for Type Hints

Here's how you can use type hints in Python:

1. **Variable Annotations**

```
# Explicitly declare the type of a variable
name: str = "Alice"
age: int = 30
is_student: bool = True
```

2. **Function Arguments and Return Types**

```
# Annotate function arguments and return types
def greet(name: str) -> str:
    return f"Hello, {name}"
```

3. **Collections and Containers**

Use `typing` module for complex types like lists, dictionaries, etc.:

```
from typing import List, Dict, Tuple

# A function that takes a list of integers and returns a string
def process_numbers(numbers: List[int]) -> str:
    return f"Total: {sum(numbers)}"

# A function that takes a dictionary and returns a tuple
```

```python
def process_data(data: Dict[str, int]) -> Tuple[str, int]:
    return ("Alice", data["age"])
```

4. **Optional Types**

Use `Optional` for arguments that can be `None`:

```python
from typing import Optional

def find_user(user_id: int) -> Optional[str]:
    if user_id == 1:
        return "Alice"
    return None  # Explicitly allowed by Optional[str]
```

5. **Custom Types**

You can use classes or aliases for custom types:

```python
# Using a class as a type
class User:
    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age

def get_user_info(user: User) -> str:
    return f"{user.name}, {user.age}"

# Using type aliases
Vector = List[float]

def scale_vector(v: Vector, factor: float) -> Vector:
    return [x * factor for x in v]
```

Example with Type Hints

Here's a complete example demonstrating type hints:

```python
from typing import List, Dict, Optional

def process_data(data: Dict[str, int], threshold: Optional[int] = None) -> List[str]:
    """Process a dictionary of data and return a list of keys above a threshold."""
    if threshold is None:
        threshold = 0
    return [key for key, value in data.items() if value > threshold]

# Example usage
data = {"a": 10, "b": 5, "c": 20}
result = process_data(data, threshold=15)
print(result)  # Output: ['c']
```

Static Type Checking with `mypy`

To enforce type hints, use a static type checker like `mypy`: 1. Install `mypy`: `bash pip install mypy` 2. Run `mypy` on your Python file: `bash mypy your_script.py` 3. `mypy` will report type errors, if any.

When to Use Type Hints

- **Larger Projects**: Type hints are especially useful in larger codebases with multiple contributors.
- **Libraries/Frameworks**: They help users understand how to use your code correctly.
- **Documentation**: Type hints serve as self-documenting code.

Limitations

- **Optional**: Type hints are not enforced at runtime (Python remains dynamically typed).
- **Verbose**: They can make code slightly more verbose, but the benefits often outweigh the costs.

By using type hints, you make your code more robust, readable, and maintainable. Start using them in your projects to level up your Python skills!