Code Formatting
Type Hints

•••

eidos.ai

Code Formatting

Code formatting refers to the practice of structuring code in a consistent style, making it easy to read and understand.

Benefits of Consistent Code Formatting

- Enhances code readability and maintainability.
- Facilitates collaboration by adhering to a common coding standard.
- Reduces the likelihood of syntax errors and bugs.

Type Checking

Type checking is the process of verifying the type of variables and expressions in code, usually done before runtime.

Benefits of Type Checking

- Helps catch errors early in the development process.
- Improves code quality and reliability.
- Assists in understanding code functionality and intention.

Python is flexible and dynamic, adopting good practices like code formatting and type checking is crucial for writing robust, error-free, and team-friendly code.

Python Tools

Black: Code Formatter

Why Use Black?

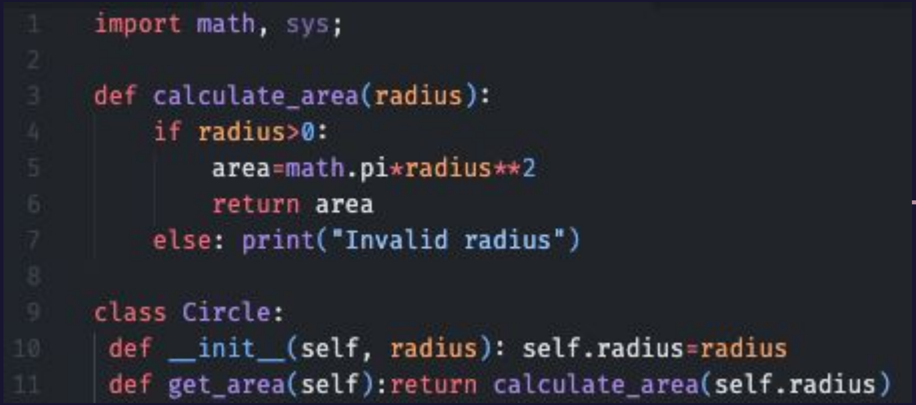
Utilizing Black as a code formatter enhances the readability and consistency of Python code, streamlining project collaboration and maintenance by automatically standardizing coding styles across different projects.

Mypy: Optional Static Typing for Python

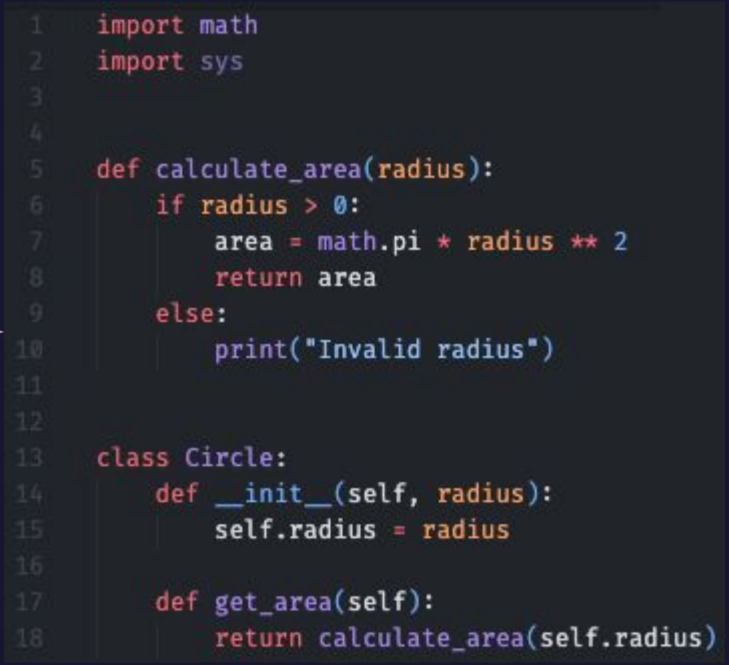
Why Use Mypy?

Implementing Mypy in Python development significantly improves code quality by enabling early detection of type-related errors, ensuring type consistency, and enhancing overall code clarity, which is especially beneficial in large-scale or team-based projects.

Black: Before and After

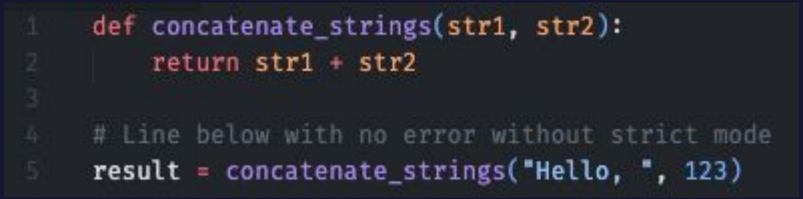


```python
import math, sys;

def calculate_area(radius):
    if radius>0:
        area=math.pi*radius**2
        return area
    else: print("Invalid radius")

class Circle:
 def __init__(self, radius): self.radius=radius
 def get_area(self):return calculate_area(self.radius)
```

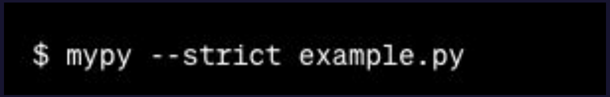


```python
import math
import sys


def calculate_area(radius):
    if radius > 0:
        area = math.pi * radius ** 2
        return area
    else:
        print("Invalid radius")


class Circle:
    def __init__(self, radius):
        self.radius = radius

    def get_area(self):
        return calculate_area(self.radius)
```

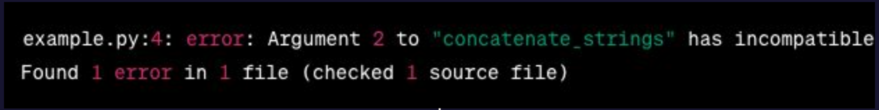
Mypy with Strict Mode: Enhanced Type Checking



```python
def concatenate_strings(str1, str2):
    return str1 + str2

# Line below with no error without strict mode
result = concatenate_strings("Hello, ", 123)
```

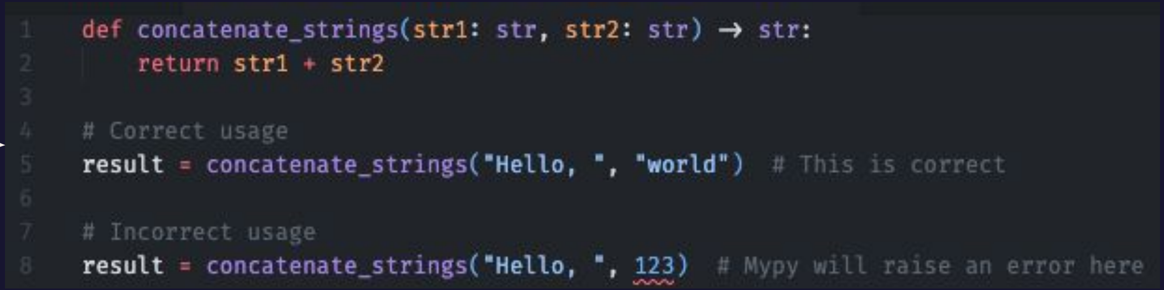


```
$ mypy --strict example.py
```



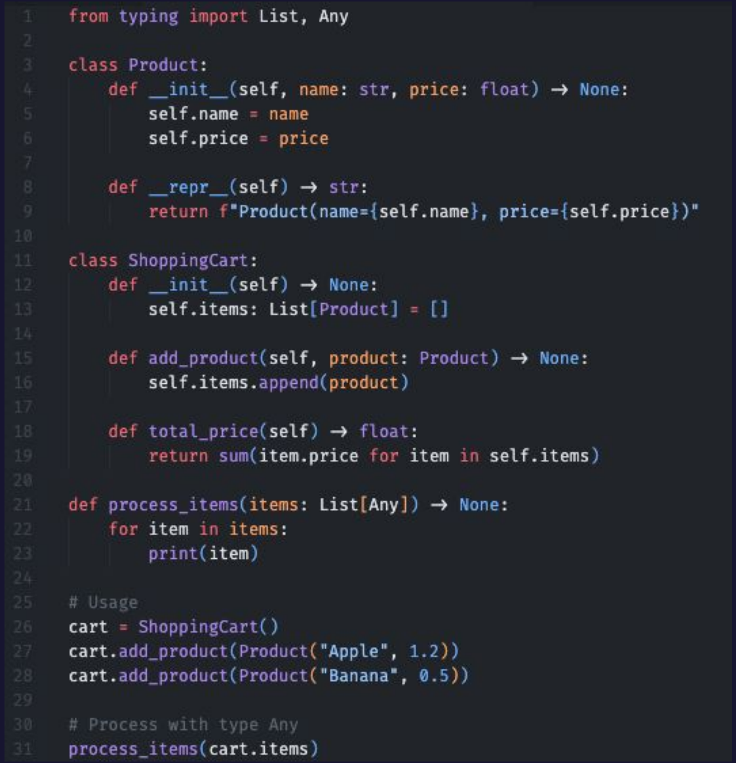
```
example.py:4: error: Argument 2 to "concatenate_strings" has incompatible
Found 1 error in 1 file (checked 1 source file)
```

- In strict mode, Mypy is more rigorous, enforcing strict type annotations.
- In the example, Mypy catches the error where an integer is passed instead of a string.
- Strict mode ensures that all function parameters and return types are explicitly annotated, leading to clearer and more reliable code.



```python
def concatenate_strings(str1: str, str2: str) -> str:
    return str1 + str2

# Correct usage
result = concatenate_strings("Hello, ", "world")  # This is correct

# Incorrect usage
result = concatenate_strings("Hello, ", 123)  # Mypy will raise an error here
```

Example with Custom Classes and Advanced Type Annotations



```python
from typing import List, Any

class Product:
    def __init__(self, name: str, price: float) -> None:
        self.name = name
        self.price = price

    def __repr__(self) -> str:
        return f"Product(name={self.name}, price={self.price})"

class ShoppingCart:
    def __init__(self) -> None:
        self.items: List[Product] = []

    def add_product(self, product: Product) -> None:
        self.items.append(product)

    def total_price(self) -> float:
        return sum(item.price for item in self.items)

def process_items(items: List[Any]) -> None:
    for item in items:
        print(item)

# Usage
cart = ShoppingCart()
cart.add_product(Product("Apple", 1.2))
cart.add_product(Product("Banana", 0.5))

# Process with type Any
process_items(cart.items)
```

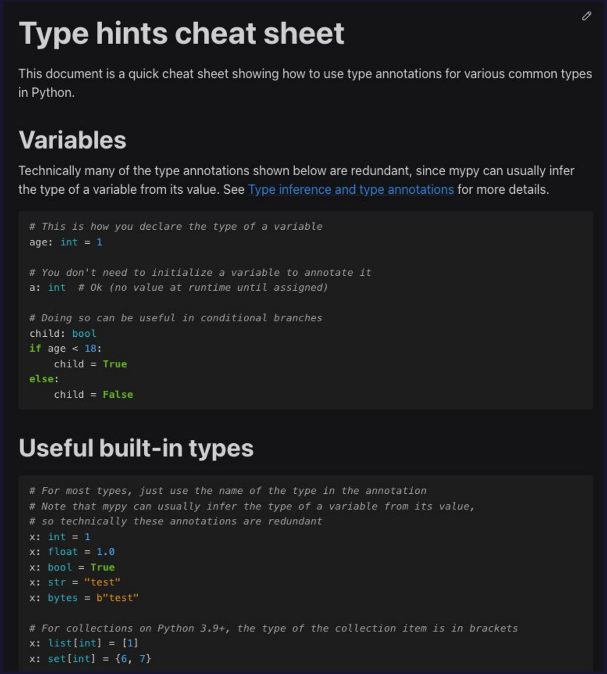
Any

Use: Any is used when the type is unknown or can vary, like in dynamic code or with external libraries without type hints.

Caution: While flexible, Any should be used sparingly as it bypasses the advantages of static type checking, potentially leading to hidden errors.

Ignore files or lines

Use: #type: ignore



Type hints cheat sheet

This document is a quick cheat sheet showing how to use type annotations for various common types in Python.

Variables

Technically many of the type annotations shown below are redundant, since mypy can usually infer the type of a variable from its value. See Type inference and type annotations for more details.

```
# This is how you declare the type of a variable
age: int = 1

# You don't need to initialize a variable to annotate it
a: int  # Ok (no value at runtime until assigned)

# Doing so can be useful in conditional branches
child: bool
if age < 18:
    child = True
else:
    child = False
```

Useful built-in types

```
# For most types, just use the name of the type in the annotation
# Note that mypy can usually infer the type of a variable from its value,
# so technically these annotations are redundant
x: int = 1
x: float = 1.0
x: bool = True
x: str = "test"
x: bytes = b"test"

# For collections on Python 3.9+, the type of the collection item is in brackets
x: list[int] = [1]
x: set[int] = {6, 7}
```

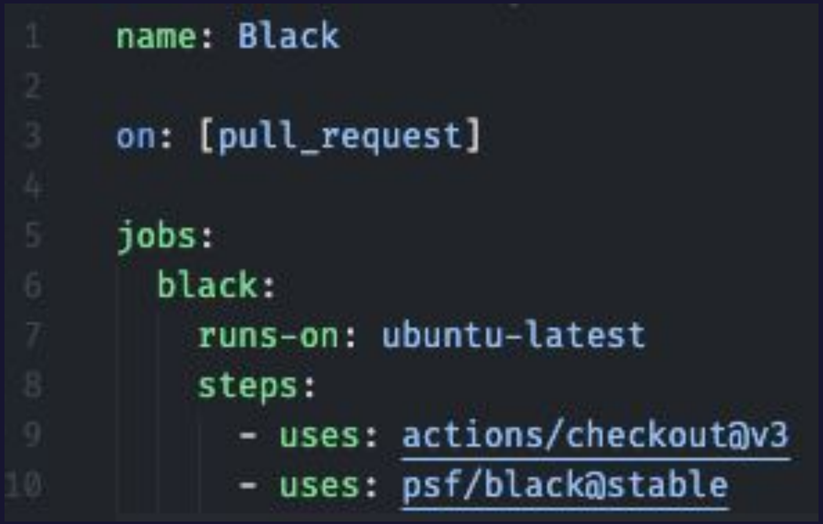
What are Stubs?

Stubs are essential in adding type hints to unannotated Python code, especially for external libraries. When stubs are missing, you can either look for existing ones, create your own, use inline typing, or employ the Any type as needed.

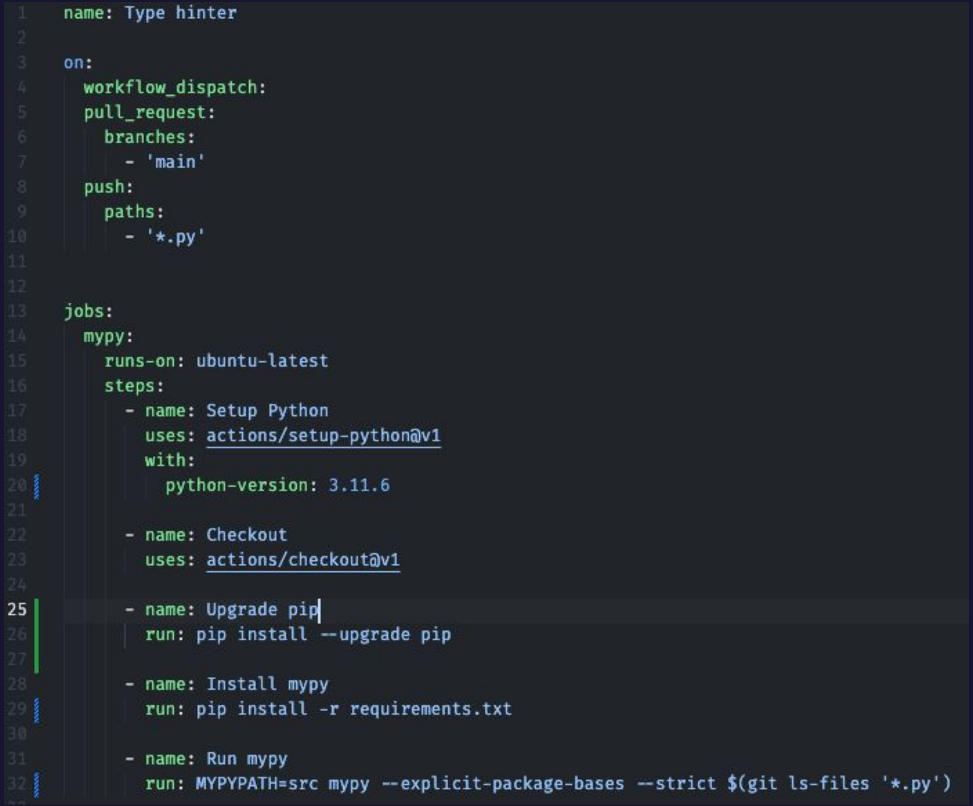
https://mypy.readthedocs.io/en/stable/

Automating Code Checks Using GitHub Actions

GitHub Actions offers a powerful platform for automating various development workflows, including automatic code formatting with Black and type checking with Mypy.



```yaml
name: Black

on: [pull_request]

jobs:
  black:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: psf/black@stable
```



```yaml
name: Type hinter

on:
  workflow_dispatch:
  pull_request:
    branches:
      - 'main'
  push:
    paths:
      - '*.py'


jobs:
  mypy:
    runs-on: ubuntu-latest
    steps:
      - name: Setup Python
        uses: actions/setup-python@v1
        with:
          python-version: 3.11.6

      - name: Checkout
        uses: actions/checkout@v1

      - name: Upgrade pip
        run: pip install --upgrade pip

      - name: Install mypy
        run: pip install -r requirements.txt

      - name: Run mypy
        run: MYPYPATH=src mypy --explicit-package-bases --strict $(git ls-files '*.py')
```