BULLETPROOF PYTHON

WRITING FEWER TESTS WITH A TYPED CODE BASE

Michael Seifert

```python
def prepend_foo(s):
    return f"foo{s}"
```

Speaker notes

A function can easily be used the wrong way, especially if it's poorly documented.

```
def prepend_foo(s):
    return f"foo{s}"
```

```
prepend_foo("bar")
"foobar"
```

Speaker notes

A function can easily be used the wrong way, especially if it's poorly documented.

```
def prepend_foo(s):
    return f"foo{s}"
```

```
prepend_foo("bar")
"foobar"
```

```
prepend_foo(b"bar")
"foob'bar'"
```

Speaker notes

A function can easily be used the wrong way, especially if it's poorly documented.

```
def prepend_foo(s):
    return f"foo{s}"
```

```
prepend_foo("bar")
"foobar"
```

```
prepend_foo(b"bar")
"foob'bar'"
```

```
prepend_foo(None)
"fooNone"
```

Speaker notes

A function can easily be used the wrong way, especially if it's poorly documented.

```python
def prepend_foo(s):
    if not isinstance(s, str):
        raise ValueError(f"Expected str, but got {type(s)}")
    return f"foo{s}"
```

Speaker notes

Checking the input type is not very Pythonic

```python
def prepend_foo(s):
    if not isinstance(s, str):
        raise ValueError(f"Expected str, but got {type(s)}")
    return f"foo{s}"
```

```python
def test_prepend_foo_raises_when_argument_is_bytes():
    with pytest.raises(ValueError):
        prepend_foo(b"bar")
```

Speaker notes

Checking the input type is not very Pythonic

TYPE ANNOTATIONS

```python
def prepend_foo(s: str) -> str:
    return f"foo{s}"

prepend_foo("bar")
```

Speaker notes

Type annotations allow us to use type checkers for static program analysis

```python
def prepend_foo(s: str) -> str:
    return f"foo{s}"

prepend_foo("bar")
```

```
Success: no issues found in 1 source file
```

Speaker notes

Type annotations allow us to use type checkers for static program analysis

```
1 def prepend_foo(s: str) -> str:
2     return f"foo{s}"
3
4 prepend_foo(b"bar")
```

```python
def prepend_foo(s: str) -> str:
    return f"foo{s}"

prepend_foo(b"bar")
```

```
prepend_foo.py:4: error: Argument 1 to "prepend_foo" has
  incompatible type "bytes"; expected "str"  [arg-type]
Found 1 error in 1 file (checked 1 source file)
```

- Type checkers verify your code before it runs

- Type checkers verify your code before it runs
- Type checkers eliminate some classes of bugs

MAKING ASSUMPTIONS EXPLICIT

```python
@dataclass
class User:
    name: str
    email: str
    password: str
```

Speaker notes

All fields are of type "str"

```python
@dataclass
class User:
    name: str
    email: str
    password: str
```

```python
user = User(
    name="seifertm",
    email="m.seifert@digitalernachschub.de",
    password="zmrzlina 🍨",
)
```

Speaker notes

All fields are of type "str"

```python
def send_message(to: str, message: str):
    ...
```

Speaker notes

Both calls are valid, but it's likely that send_message requires and email address as a recipient.

```python
def send_message(to: str, message: str):
    ...
```

```python
send_message(to=user.email, message="Hello!")  # valid call
```

Speaker notes

Both calls are valid, but it's likely that send_message requires and email address as a recipient.

```python
def send_message(to: str, message: str):
    ...
```

```python
send_message(to=user.email, message="Hello!")  # valid call
```

```python
send_message(to=user.name, message="Hello!")  # valid call
```

Speaker notes

Both calls are valid, but it's likely that send_message requires and email address as a recipient.

```
def send_message(to: str, message: str):
    if not is_email_address(to):
        raise ValueError()
    ...
```

Speaker notes

We can validate the input argument and write a unit test that triggers the failing validation

```python
def send_message(to: str, message: str):
    if not is_email_address(to):
        raise ValueError()
    ...
```

```python
def test_send_message_raises_when_to_is_not_an_email_address()
    ...
```

Speaker notes

We can validate the input argument and write a unit test that triggers the failing validation

```python
from typing import NewType

EmailAddress = NewType("EmailAddress", str)
```

Speaker notes

EmailAddress is a special type of string.

EmailAddress can be used anywhere, where str is used, but not the other way around.

```python
from typing import NewType

EmailAddress = NewType("EmailAddress", str)
```

```python
@dataclass
class User:
    name: str
    email: EmailAddress
    password: str
```

Speaker notes

EmailAddress is a special type of string.

EmailAddress can be used anywhere, where str is used, but not the other way around.

```python
from typing import NewType

EmailAddress = NewType("EmailAddress", str)
```

```python
@dataclass
class User:
    name: str
    email: EmailAddress
    password: str
```

```python
def send_message(to: EmailAddress, message: str):
    ...
```

Speaker notes

EmailAddress is a special type of string.

EmailAddress can be used anywhere, where str is used, but not the other way around.

```
user = User(
    name="seifertm",
    email=EmailAddress("m.seifert@digitalernachschub.de"),
    password="zmrzlina 🍨",
)
```

```
send_message(to=user.email, message="Hello!")
```

```
send_message(to=user.email, message="Hello!")
```

```
Success: no issues found in 1 source file
```

```
send_message(to=user.email, message="Hello!")
```

```
Success: no issues found in 1 source file
```

```
send_message(to=user.name, message="Hello!")
```

```
send_message(to=user.email, message="Hello!")
```

```
Success: no issues found in 1 source file
```

```
send_message(to=user.name, message="Hello!")
```

```
user.py:21: error: Argument "to" to "send_message" has
  incompatible type "str"; expected "EmailAddress"  [arg-type]
Found 1 error in 1 file (checked 1 source file)
```

```python
def save_to_database(user: User):
    query = """
        INSERT INTO Users(name, email, password)
        VALUES (:name, :email, :password)
    """
    database.execute(query, **asdict(user))
```

Speaker notes

A users plaintext password was just stored in the database.

```
def save_to_database(user: User):
    query = """
        INSERT INTO Users(name, email, password)
        VALUES (:name, :email, :password)
    """
    database.execute(query, **asdict(user))
```

```
save_to_database(user)
```

Speaker notes

A users plaintext password was just stored in the database.

```python
def save_to_database(user: User):
    query = """
        INSERT INTO Users(name, email, password)
        VALUES (:name, :email, :password)
    """
    database.execute(query, **asdict(user))
```

```python
save_to_database(user)
```

Speaker notes

A users plaintext password was just stored in the database.

```
HashedPassword = NewType("HashedPassword", str)
```

Speaker notes

Requiring a HashedPassword for User.password forces users to looks for ways to provide a HashedPassword when initializing a User object.

```
HashedPassword = NewType("HashedPassword", str)
```

```
@dataclass
class User:
    name: str
    email: EmailAddress
    password: HashedPassword
```

Speaker notes

Requiring a HashedPassword for User.password forces users to looks for ways to provide a HashedPassword when initializing a User object.

```python
HashedPassword = NewType("HashedPassword", str)
```

```python
@dataclass
class User:
    name: str
    email: EmailAddress
    password: HashedPassword
```

```python
user = User(
    name="seifertm",
    email=EmailAddress("m.seifert@digitalernachschub.de"),
    password="zmrzlina 🍨",
)
```

Speaker notes

Requiring a HashedPassword for User.password forces users to looks for ways to provide a HashedPassword when initializing a User object.

```python
HashedPassword = NewType("HashedPassword", str)
```

```python
@dataclass
class User:
    name: str
    email: EmailAddress
    password: HashedPassword
```

```python
user = User(
    name="seifertm",
    email=EmailAddress("m.seifert@digitalernachschub.de"),
    password="zmrzlina 🍨",
)
```

```
user.py:16: error: Argument "password" to "User" has
  incompatible type "str"; expected "HashedPassword" [arg-type
Found 1 error in 1 file (checked 1 source file)
```

Speaker notes

Requiring a HashedPassword for User.password forces users to looks for ways to provide a HashedPassword when initializing a User object.

TAKE AWAYS

- NewType does not perform validation

TAKE AWAYS

- NewType does not perform validation
- NewType makes a type more specific

TAKE AWAYS

- NewType does not perform validation
- NewType makes a type more specific
- More specific types narrow down use cases

TAKE AWAYS

- NewType does not perform validation
- NewType makes a type more specific
- More specific types narrow down use cases
- As a result, the number of required test cases is reduced

DEPENDENT TYPES

```
class Point1D:
    x: float
```

Speaker notes

A type represents a set of values.

```
class Point1D:
    x: float
```

```
class Point2D:
    x: float
    y: float
```

Speaker notes

A type represents a set of values.

```
class Point1D:
    x: float
```

```
class Point2D:
    x: float
    y: float
```

```
...
```

Speaker notes

A type represents a set of values.

```python
# ⚠️ Pseudo Python, don't try this at home
class PointND(N: int):
    ...
```

Speaker notes

A dependent type represents a family of types whose exact type changes based on a value.

Functions that return dependent types are called dependent functions.

```python
# ⚠️ Pseudo Python, don't try this at home
class PointND(N: int):
    ...
```

```python
# ⚠️ Pseudo Python, don't try this at home
def random_point(dimensions: int) -> PointND[dimensions]:
    ...
```

Speaker notes

A dependent type represents a family of types whose exact type changes based on a value.

Functions that return dependent types are called dependent functions.

```
def read_file(
    path: Path, mode: Literal["r", "rb"]
) -> str | bytes:
    ...
```

Speaker notes

The function signature doesn't say that mode="r" cannot return a "bytes" result.

```python
def read_file(
    path: Path, mode: Literal["r", "rb"]
) -> str | bytes:
    ...
```

```python
def test_read_file_returns_str_when_mode_is_r():
    ...
```

Speaker notes

The function signature doesn't say that mode="r" cannot return a "bytes" result.

```
def read_file(
    path: Path, mode: Literal["r", "rb"]
) -> str | bytes:
    ...
```

```
def test_read_file_returns_str_when_mode_is_r():
    ...
```

```
def test_read_file_returns_bytes_when_mode_is_rb():
    ...
```

Speaker notes

The function signature doesn't say that mode="r" cannot return a "bytes" result.

```python
from typing import overload

@overload
def read_file(path: Path, mode: Literal["r"]) -> str:
    ...
```

Speaker notes

Overload allows us to specify multiple functions signatures.

```python
from typing import overload

@overload
def read_file(path: Path, mode: Literal["r"]) -> str:
    ...
```

```python
@overload
def read_file(path: Path, mode: Literal["rb"]) -> bytes:
    ...
```

Speaker notes

Overload allows us to specify multiple functions signatures.

```python
from typing import overload

@overload
def read_file(path: Path, mode: Literal["r"]) -> str:
    ...
```

```python
@overload
def read_file(path: Path, mode: Literal["rb"]) -> bytes:
    ...
```

```python
def read_file(path, mode):
    # Implementation goes here
```

Speaker notes

Overload allows us to specify multiple functions signatures.

TAKE AWAYS

- *overload* makes ambiguous function signatures more specific

TAKE AWAYS

- *overload* makes ambiguous function signatures more specific
- *overload* makes dependencies between function arguments and outputs visible

EXHAUSTIVENESS CHECKING

```
class Character(Enum):
    Alice = auto()
    Bob = auto()
```

Speaker notes

ValueError only discovered if there's a test case that triggers it.

```python
class Character(Enum):
    Alice = auto()
    Bob = auto()
```

```python
def draw(character: Character):
    match character:
        case Character.Alice:
            ...
        case Character.Bob:
            ...
        case _:
            raise ValueError("This should never happen")
```

Speaker notes

ValueError only discovered if there's a test case that triggers it.

```
class Character(Enum):
    Alice = auto()
    Bob = auto()
    Eve = auto()
```

Speaker notes

After extending the enum, we end up with a missing "case" in the match statement of the draw function. This is only uncovered if we have a test that passes the new value to the draw function.

```
class Character(Enum):
    Alice = auto()
    Bob = auto()
    Eve = auto()
```

```
draw(Character.Eve)
```

Speaker notes

After extending the enum, we end up with a missing "case" in the match statement of the draw function. This is only uncovered if we have a test that passes the new value to the draw function.

```
class Character(Enum):
    Alice = auto()
    Bob = auto()
    Eve = auto()
```

```
draw(Character.Eve)
```

```
def test_draw_raises_when_enum_value_is_eve():
    ...
```

Speaker notes

After extending the enum, we end up with a missing "case" in the match statement of the draw function. This is only uncovered if we have a test that passes the new value to the draw function.

```python
from typing import assert_never

def draw(character: Character):
    match character:
        case Character.Alice:
            ...
        case Character.Bob:
            ...
        case _ as impossible:
            assert_never(impossible)
```

Speaker notes

assert_never casues the type checker to report an error when the statement is reachable

```python
from typing import assert_never

def draw(character: Character):
    match character:
        case Character.Alice:
            ...
        case Character.Bob:
            ...
        case _ as impossible:
            assert_never(impossible)
```

```python
draw(Character.Eve)
```

Speaker notes

assert_never casues the type checker to report an error when the statement is reachable

```python
from typing import assert_never

def draw(character: Character):
    match character:
        case Character.Alice:
            ...
        case Character.Bob:
            ...
        case _ as impossible:
            assert_never(impossible)
```

```python
draw(Character.Eve)
```

```
characters.py:16: error: Argument 1 to "assert_never" has
  incompatible type "Literal[Character.Eve]";
  expected "NoReturn"  [arg-type]
Found 1 error in 1 file (checked 1 source file)
```

Speaker notes

assert_never casues the type checker to report an error when the statement is reachable

TAKE AWAYS

- *assert_never* causes the type checker to report an error when the assert_never statement is reachable.

SUMMARY

- *NewType* can make assumptions visible

SUMMARY

- *NewType* can make assumptions visible
- *NewType* can prevent security issues

SUMMARY

- *NewType* can make assumptions visible
- *NewType* can prevent security issues
- *overload* helps making function signatures more specific

SUMMARY

- *NewType* can make assumptions visible
- *NewType* can prevent security issues
- *overload* helps making function signatures more specific
- *assert_never* can uncover code branches that are unaccounted for

There are two ways to write error-free programs; only the third one works.

—Alan J. Perlis, Epigrams in Programming (1982)

There are two ways to write error-free programs; only the third one works.

—Alan J. Perlis, Epigrams in Programming (1982)

www.seifertm.de

LinkedIn, GitHub: @seifertm