Type-Checked Python

Carl Meyer

INSTAGRAM

@carljm

@carljm

python™

@carljm

python™

Instagram

The Plan

1. **WHY** type
2. **HOW** to even type
3. **GRADUAL** typing
4. **ISSUES** you may run into

WHY add annotations to your Python?

"Items"?

```
def process(self, items):
    for item in items:
        self.append(item.value.id)
```

"Items"?

```
def process(self, items):
    for item in items:
        self.append(item.value.id)
```

"Items"?

```
def process(self, items):
    for item in items:
        self.append(item.value.id)
```

"Items"?

```
def process(self, items):
    for item in items:
        self.append(item.value.id)
```

"Items"?

```
def process(self, items):
    for item in items:
        self.append(item.value.id)
```

"Items"?

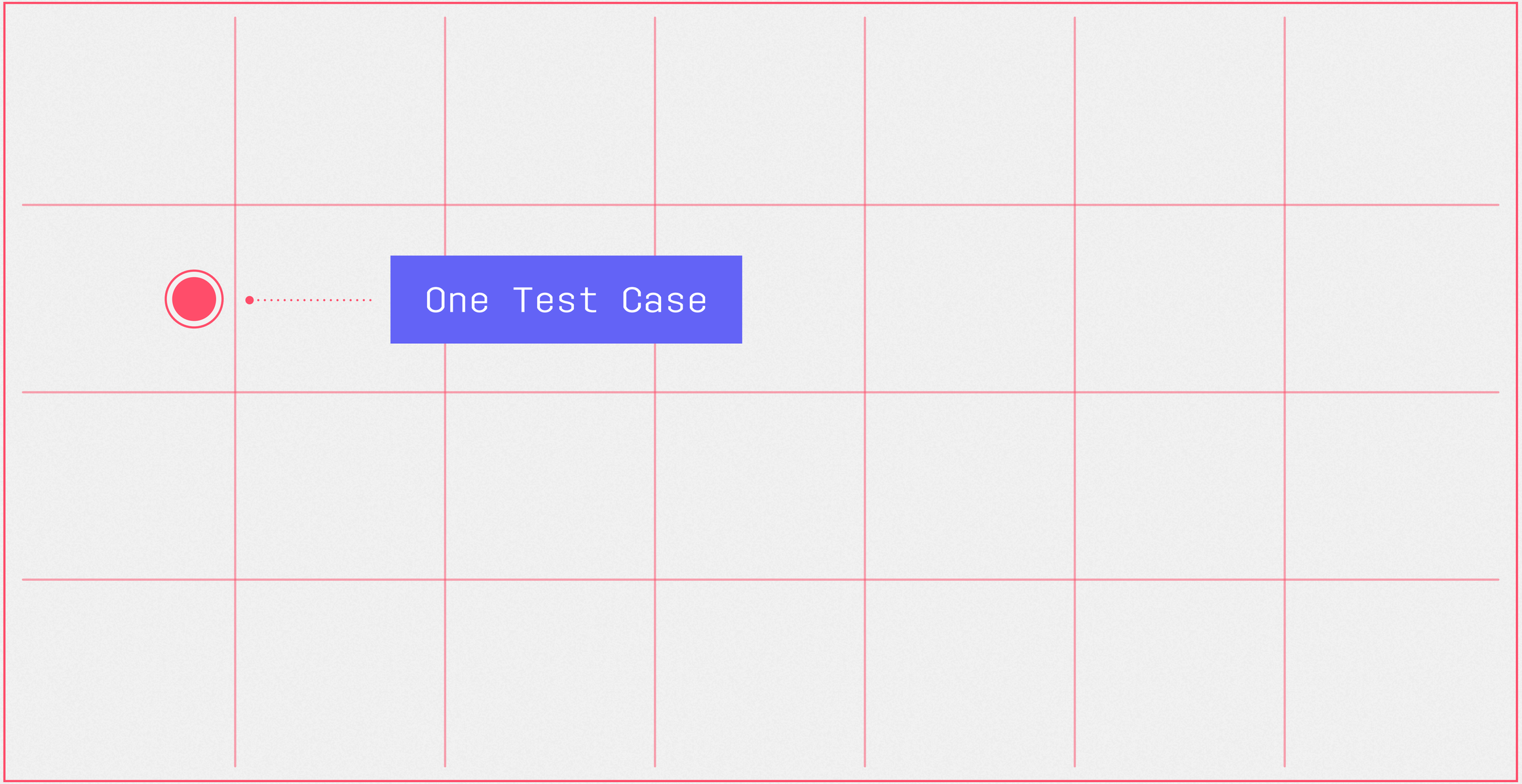
```
from typing import Sequence
from .models import Item

def process(self, items: Sequence[Item]) -> None:
    for item in items:
        self.append(item.value.id)
```

"Items"?

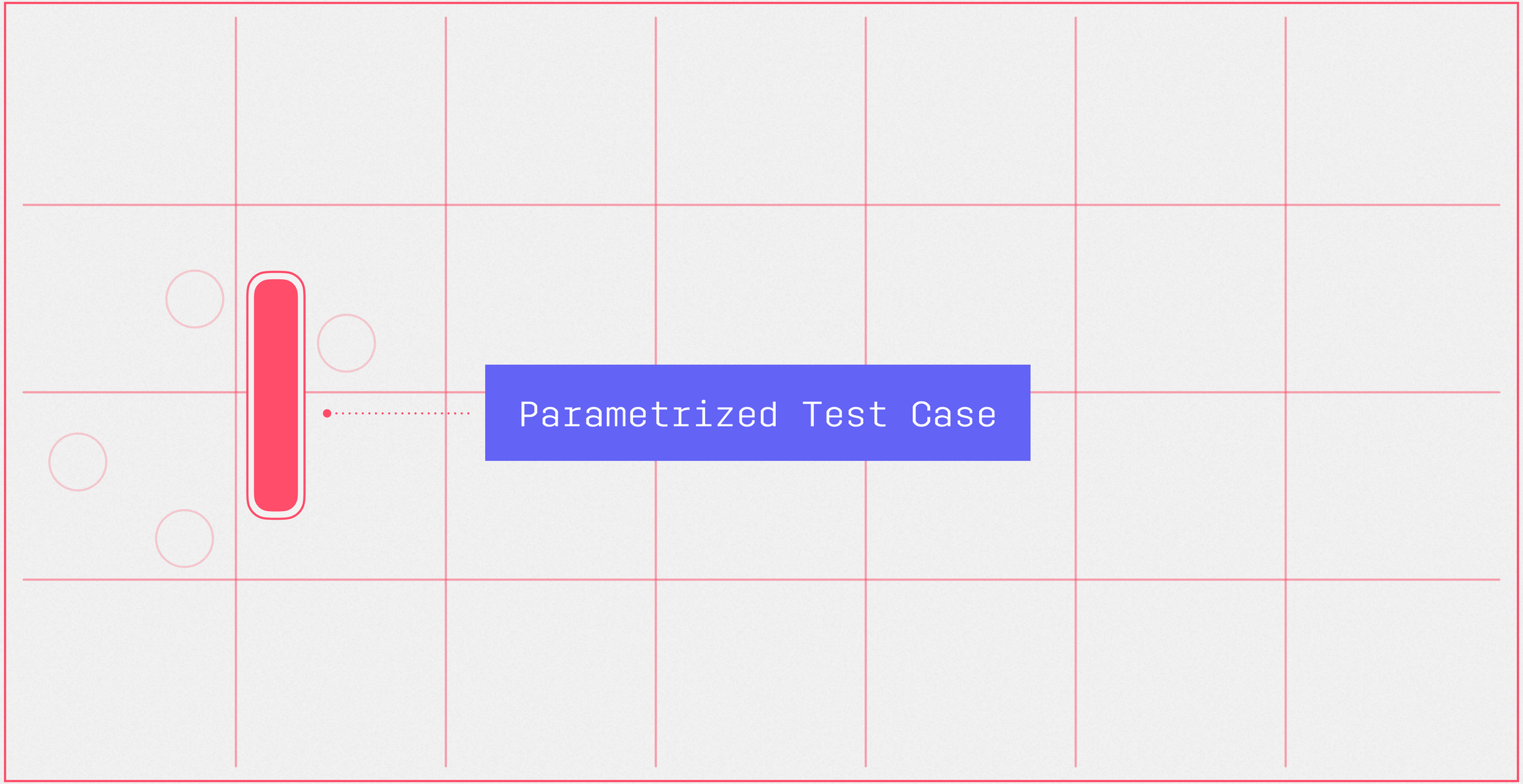
```
from typing import Sequence
from .models import Item

def process(self, items: Sequence[Item]) -> None:
    for item in items:
        self.append(item.value.id)
```

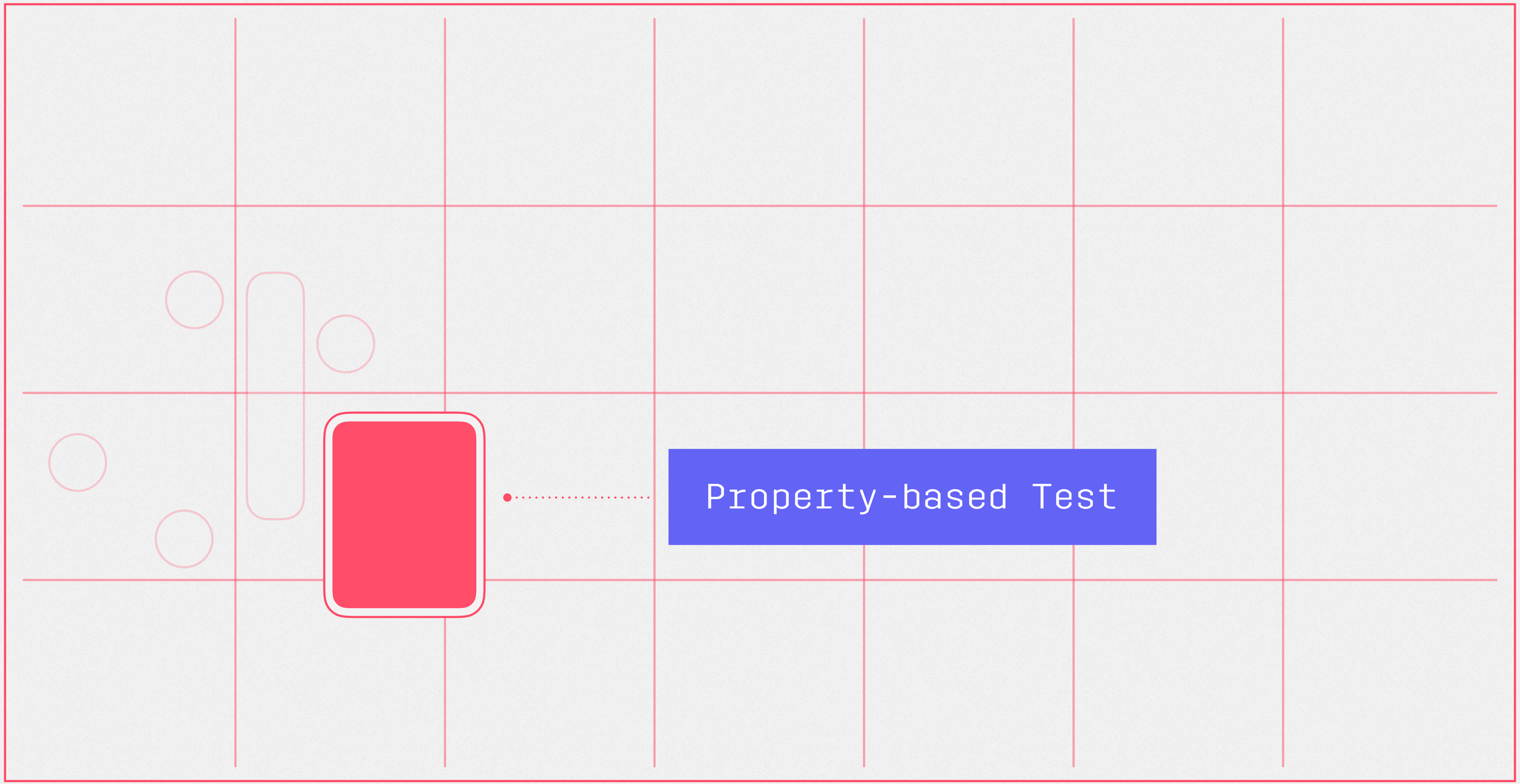
"That's cool, but I don't need it;
I'd catch that with a test!"

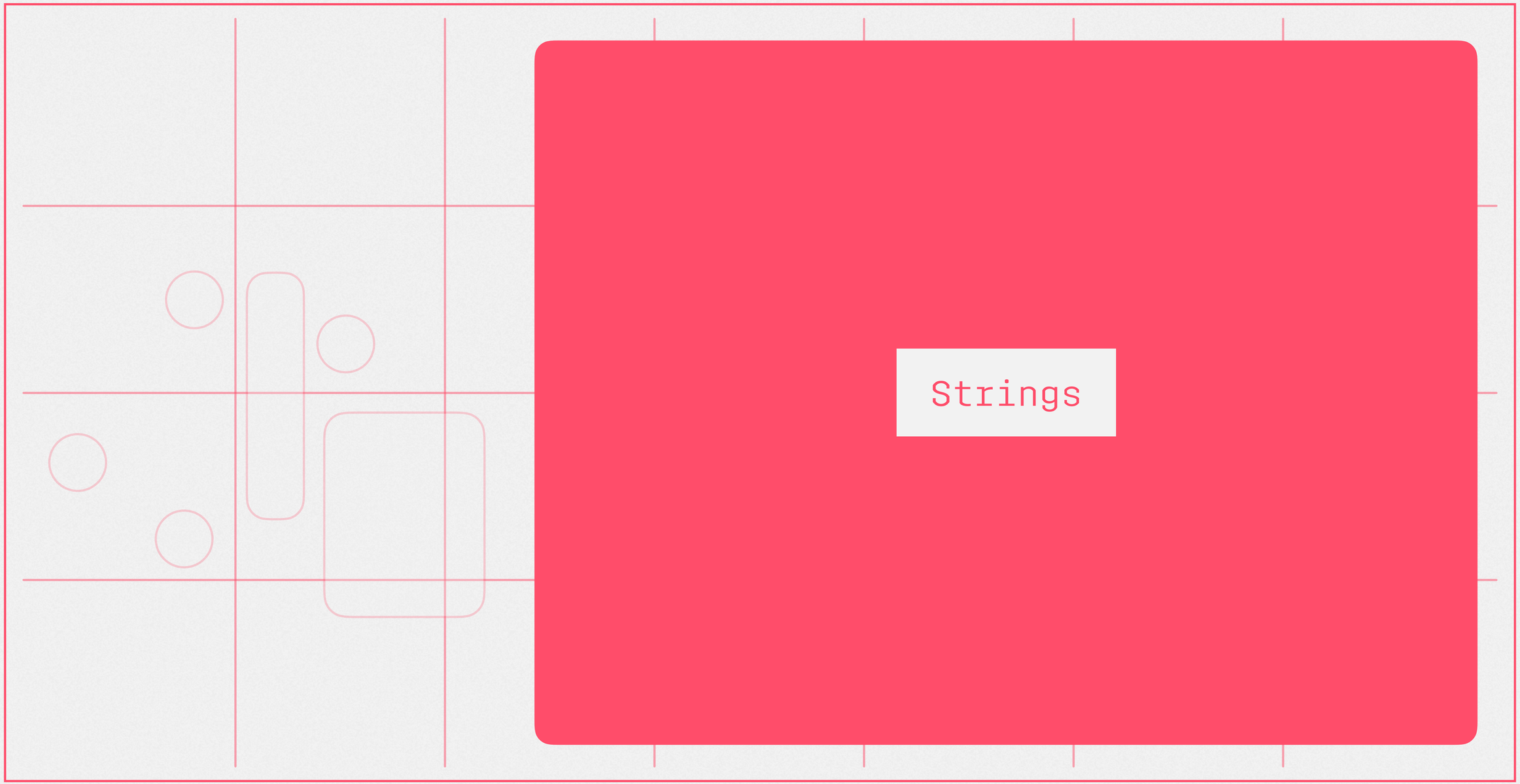
— PYTHONISTA

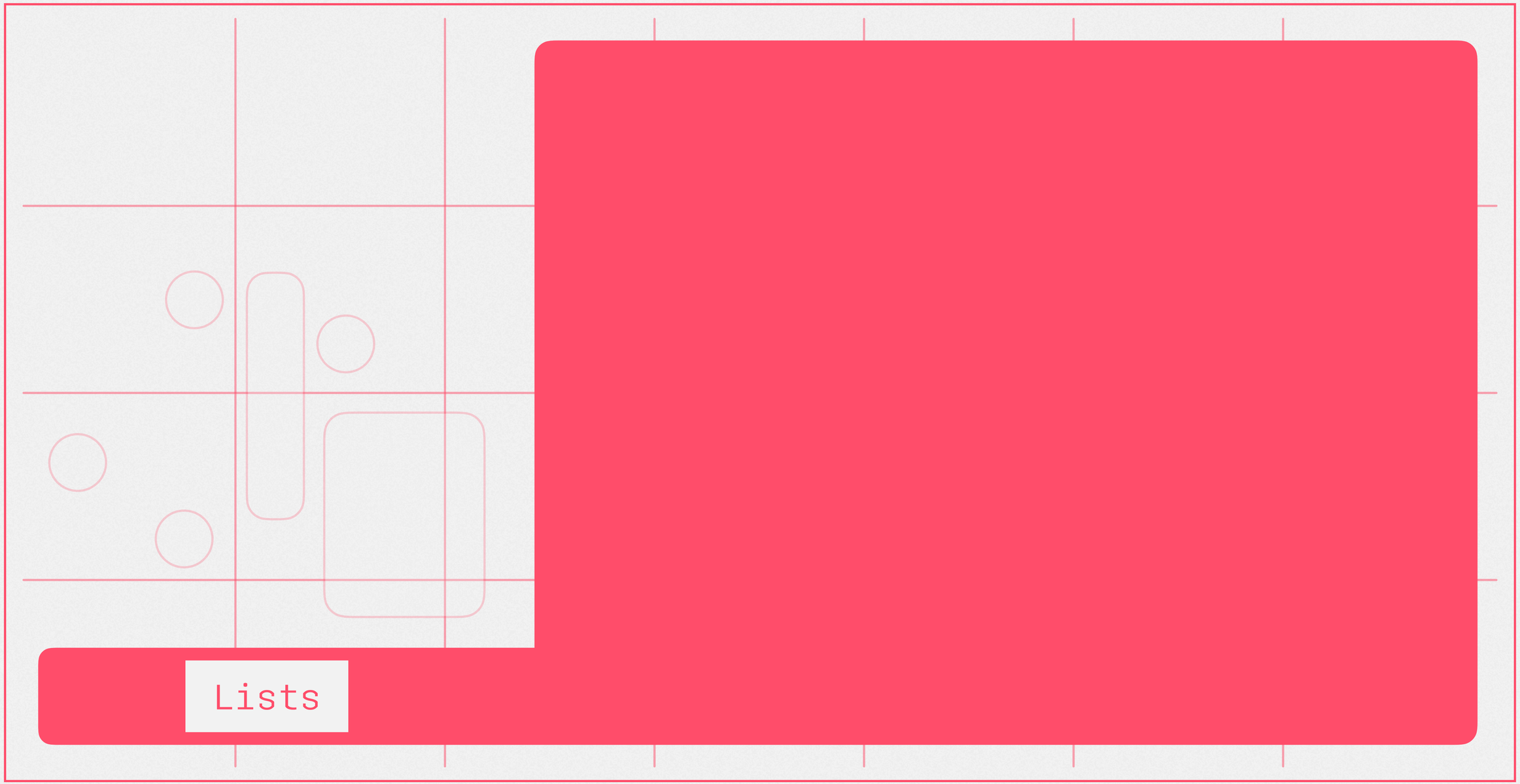

One Test Case


More Test Cases

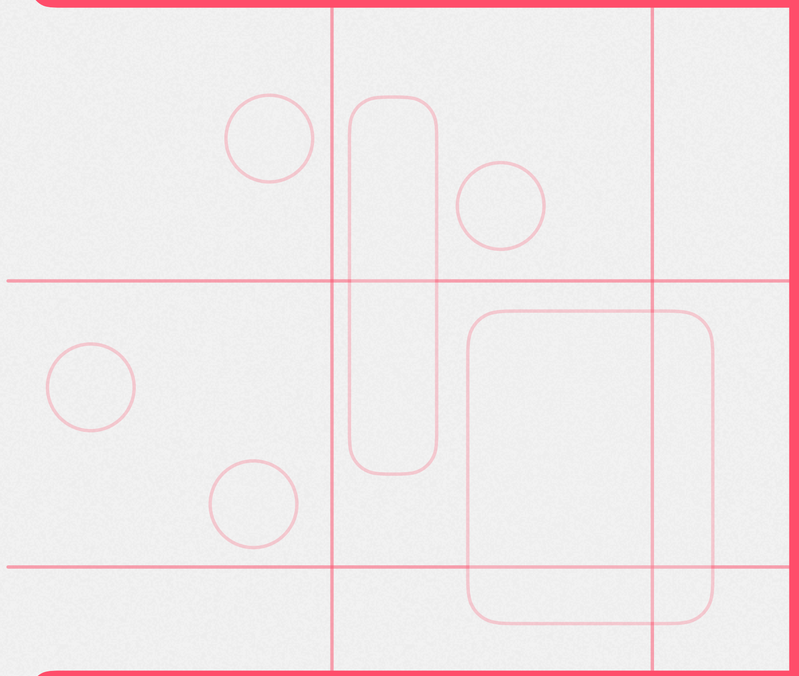

Parametrized Test Case

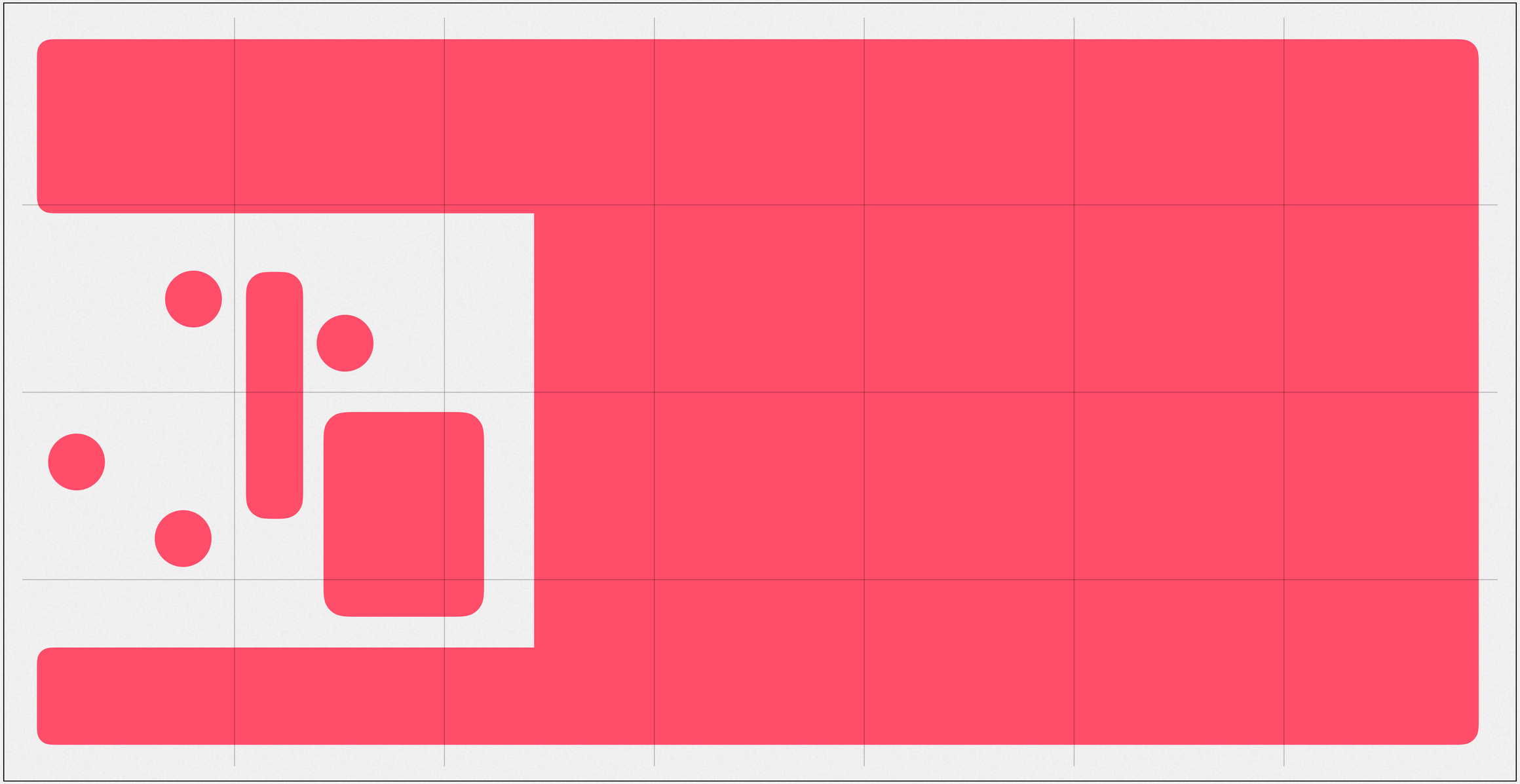

Property-based Test


Strings


Lists

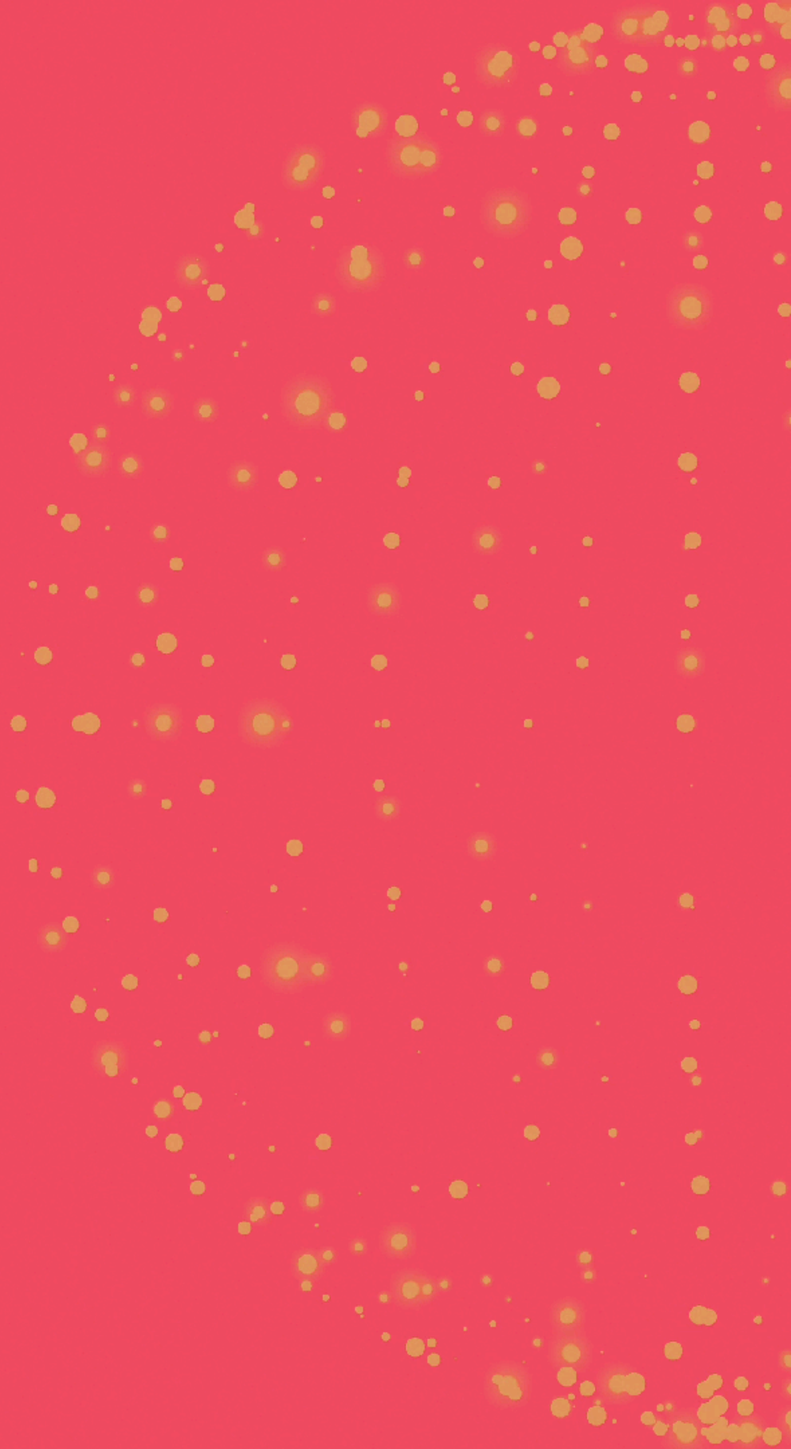
Dictionaries





The Plan

1. **WHY** type
2. **HOW** to even type
3. **GRADUAL** typing
4. **ISSUES** you may run into



HOW to even type

square.py

```
def square(x: int) -> int:
    return x**2
```

square.py

```
def square(x: int) -> int:
    return x**2
```

square.py

```
def square(x: int) -> int:
    return x**2

square(3)
square('foo')
square(4) + 'foo'
```

Running Mypy

```
$ pip install mypy
$ mypy square.py

square.py:5: error: Argument 1 to "square"
  has incompatible type "str"; expected "int"

square.py:6: error: Unsupported operand types
  for + ("int" and "str")
```

square.py

```
def square(x: int) -> int:
    return x**2


square(3)
square('foo')
square(4) + 'foo'
```

```
Argument 1 to "square" has incompatible type
"str"; expected "int"
```

square.py

```
def square(x: int) -> int:
    return x**2


square(3)
square('foo')
square(4) + 'foo'
```

```
Unsupported operand types for + ("int" and
"str")
```

Type Inference

```
from typing import Tuple

class Photo:
    def __init__(self, width: int, height: int) -> None:
        self.width = width
        self.height = height

    def get_dimensions(self) -> Tuple[str, str]:
        return (self.width, self.height)
```

Type Inference

```
from typing import Tuple

class Photo:
    def __init__(self, width: int, height: int) -> None:
        self.width = width
        self.height = height

    def get_dimensions(self) -> Tuple[str, str]:
        return (self.width, self.height)
```

```
Incompatible return value type (got "Tuple[int, int]",
expected "Tuple[str, str]")
```

Type Inference

```
photos = [
    Photo(640, 480),
    Photo(1024, 768),
]

photos.append('foo')
```

Type Inference

```
photos = [
    Photo(640, 480),
    Photo(1024, 768),
]

photos.append('foo')
```

```
Argument 1 to "append" of "list" has incompatible type
"str"; expected "Photo"
```

Variable Annotation

```
class Photo:
    def __init__(self, width: int, height: int) -> None:
        self.width = width
        self.height = height
        self.tags = []
```

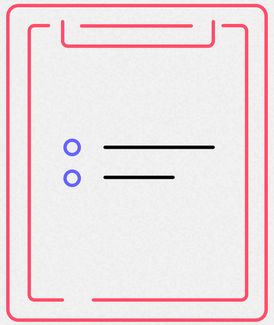
Variable Annotation

```
class Photo:
    def __init__(self, width: int, height: int) -> None:
        self.width = width
        self.height = height
        self.tags = []
```

```
Need type annotation for variable
```

Variable Annotation

```
from typing import List

class Photo:
    def __init__(self, width: int, height: int) -> None:
        self.width = width
        self.height = height
        self.tags: List[str] = []
```



Review

Annotate Function Signatures

ARGUMENTS AND RETURN VALUES

Annotate Variables

ONLY IF YOU HAVE TO

Union

```
from typing import Union

def get_foo_or_bar(id: int) -> Union[Foo, Bar]:
    ...

def get_foo_or_none(id: int) -> Union[Foo, None]:
    ...
```

Union

```
from typing import Union, Optional

def get_foo_or_bar(id: int) -> Union[Foo, Bar]:
    ...

def get_foo_or_none(id: int) -> Union[Foo, None]:
    ...

def get_foo_or_none(id: int) -> Optional[Foo]:
    ...
```

Optional

```
from typing import Optional

def get_foo(foo_id: Optional[int]) -> Optional[Foo]:
    if foo_id is None:
        return None
    return Foo(foo_id)

my_foo = get_foo(3)
my_foo.id  # error: NoneType has no attribute 'id'
```

Function Overloads

```
from typing import Optional, overload

@overload
def get_foo(foo_id: None) -> None:
    pass

@overload
def get_foo(foo_id: int) -> Foo:
    pass

def get_foo(foo_id: Optional[int]) -> Optional[Foo]:
    if foo_id is None:
        return None
    return Foo(foo_id)

reveal_type(get_foo(None))   # None
reveal_type(get_foo(1))      # Foo
```

Function Overloads

```
from typing import Optional, overload

@overload
def get_foo(foo_id: None) -> None:
    pass

@overload
def get_foo(foo_id: int) -> Foo:
    pass

def get_foo(foo_id: Optional[int]) -> Optional[Foo]:
    if foo_id is None:
        return None
    return Foo(foo_id)

reveal_type(get_foo(None))   # None
reveal_type(get_foo(1))      # Foo
```

Generics

```
numbers: List[int] = []

books_by_id: Dict[int, Book] = {}

int_tree: TreeNode[int] = TreeNode(3)
str_tree: TreeNode[str] = TreeNode('foo')
book_tree: TreeNode[Book] = TreeNode(mybook)
```

Generics

```
from typing import Generic, TypeVar

TNodeValue = TypeVar('TNodeValue')

class TreeNode(Generic[TNodeValue]):
    def __init__(self, value: TNodeValue) -> None:
        self.value = value

node = TreeNode(3)
reveal_type(node)          # TreeNode[int]
reveal_type(node.value)    # int
```

Generics

```
from typing import Generic, TypeVar

TNodeValue = TypeVar('TNodeValue')

class TreeNode(Generic[TNodeValue]):
    def __init__(self, value: TNodeValue) -> None:
        self.value = value

node = TreeNode(3)
reveal_type(node)       # TreeNode[int]
reveal_type(node.value) # int
```

Generic Functions

```
from typing import TypeVar

AnyStr = TypeVar('AnyStr', str, bytes)

def concat(a: AnyStr, b: AnyStr) -> AnyStr:
    return a + b

concat('foo', b'bar')                    # typecheck error!
concat(3, 6)                             # typecheck error!
reveal_type(concat('foo', 'bar'))        # str
reveal_type(concat(b'foo', b'bar'))      # bytes
```

Generic Functions

```
from typing import AnyStr


def concat(a: AnyStr, b: AnyStr) -> AnyStr:
    return a + b

concat('foo', b'bar')                  # typecheck error!
concat(3, 6)                           # typecheck error!
reveal_type(concat('foo', 'bar'))      # str
reveal_type(concat(b'foo', b'bar'))    # bytes
```

Review

Annotate Function Signatures

ARGUMENTS AND RETURN VALUES

Annotate Variables

ONLY IF YOU HAVE TO

Unions and Optionals

USE SPARINGLY

Overloads and Generics

TEACH THE TYPE CHECKER TO BE SMARTER


Where's my duck?

Where's my duck?

```
def render(obj):
    return obj.render()
```

Where's my duck?

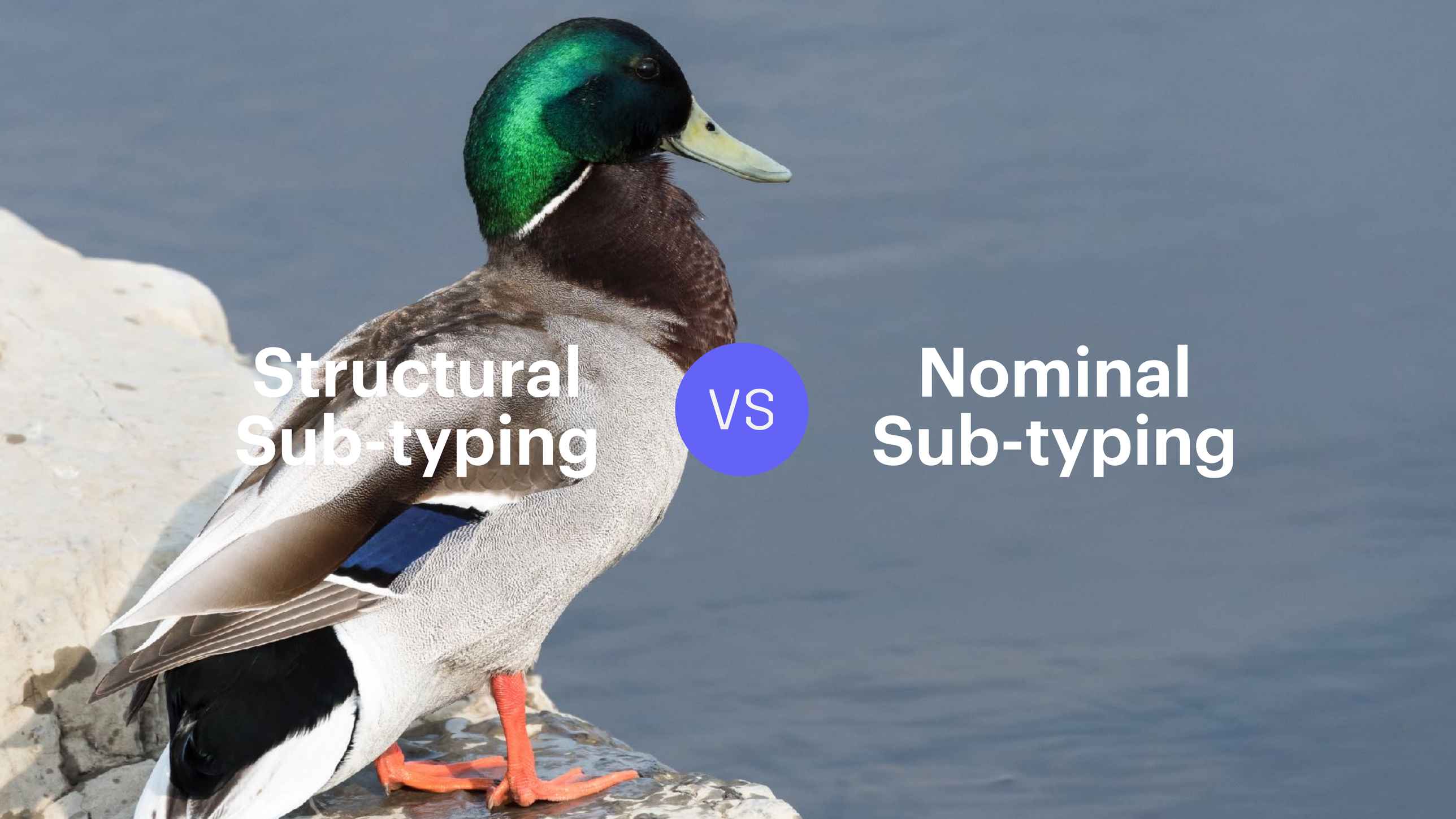
```
def render(obj: object) -> str:
    return obj.render()
```

```
"object" has no attribute "render"
```

Where's my duck?

```python
from typing import Any

def render(obj: Any) -> str:
    return obj.render()

render(3) # typechecks, but fails at runtime
```

```python
from typing_extensions import Protocol

class Renderable(Protocol):
    def render(self) -> str: ...

def render(obj: Renderable) -> str:
    return obj.render()

class Foo:
    def render(self) -> str:
        return "Foo!"

render(Foo()) # clean!
render(3)     # error: expected Renderable
```


Structural
Sub-typing
VS
Nominal
Sub-typing

Escape Hatches

Escape Hatch #1: Any

```
from typing import Any

def __getattr__(self, name: str) -> Any:
    return getattr(self.wrapped, name)

my_config: Dict[str, Union[str, int, List[Union[str,
int], Dict[str, Union[str, float]]]]]

my_config: Dict[str, Any]

# (could also use TypedDict)
```

Escape Hatch #2: cast

```
from typing import cast

my_config = get_config_var('my_config')
reveal_type(my_config)  # Any

my_config = cast(
    Dict[str, int],
    get_config_var('my_config'),
)
reveal_type(my_config)  # Dict[str, int]
```

Escape Hatch #3: ignore

```
triggers_a_mypy_bug('foo')  # type: ignore

# github.com/python/mypy/issues/1362
@property  # type: ignore
@timer
def is_visible(self) -> bool:
    ...
```

Escape Hatch #4: stub (pyi) files

```
$ ls

fastmath.so
fastmath.pyi
```

Escape Hatch #4: stub (pyi) files

```
# fastmath.pyi

def square(x: int) -> int:
    ...

class Complex:
    def __init__(real: int, imag: int) -> None:
        self.real = real
        self.imag = imag
```

Review

Annotate Function Signatures

ARGUMENTS AND RETURN VALUES

Annotate Variables

ONLY IF YOU HAVE TO

Unions and Optionals

USE SPARINGLY

Overloads and Generics

TEACH THE TYPE CHECKER TO BE SMARTER

Protocols

STATICALLY CHECKED DUCK-TYPING!

Escape Hatches

ANY, CAST, IGNORE, STUB FILES

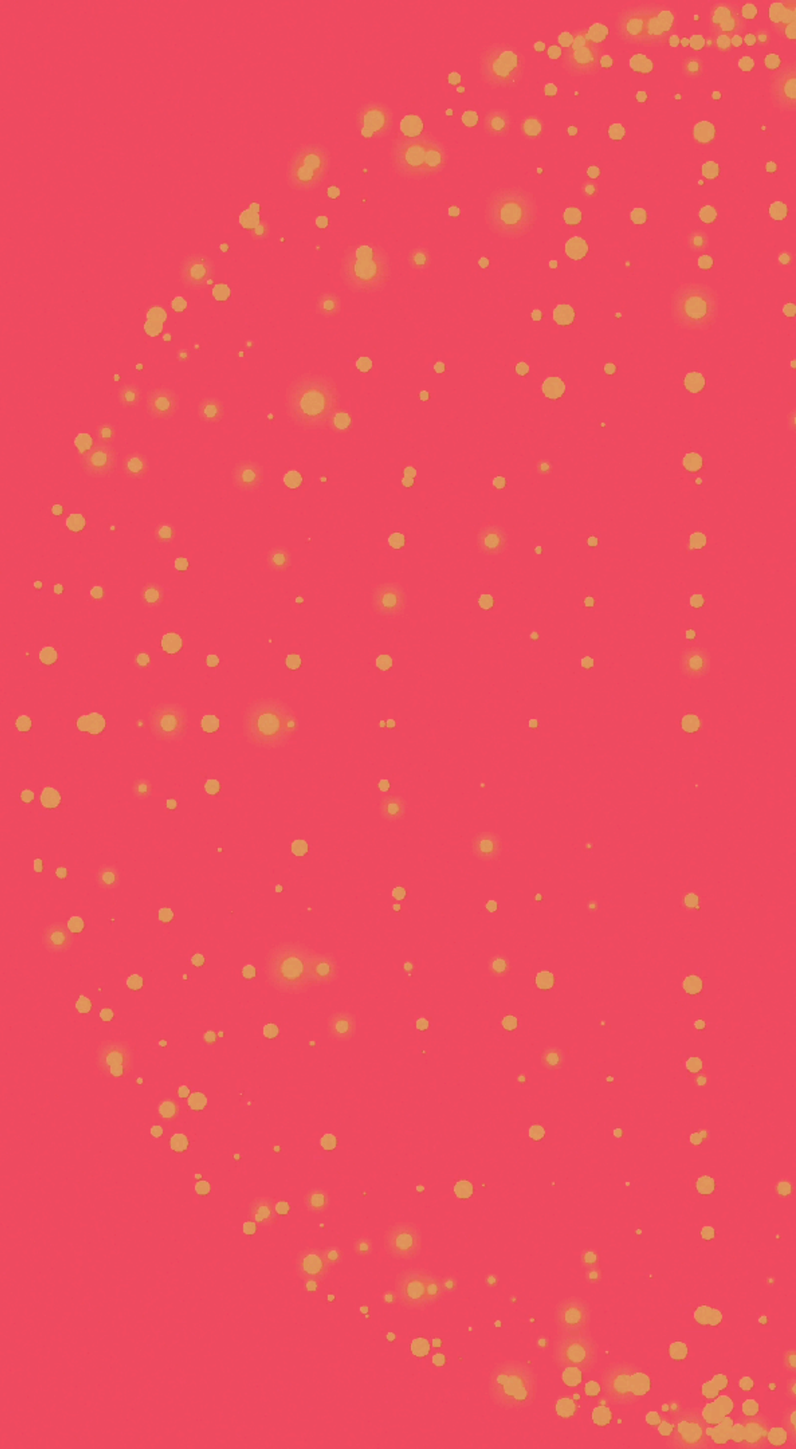
The Plan

1 **WHY** type

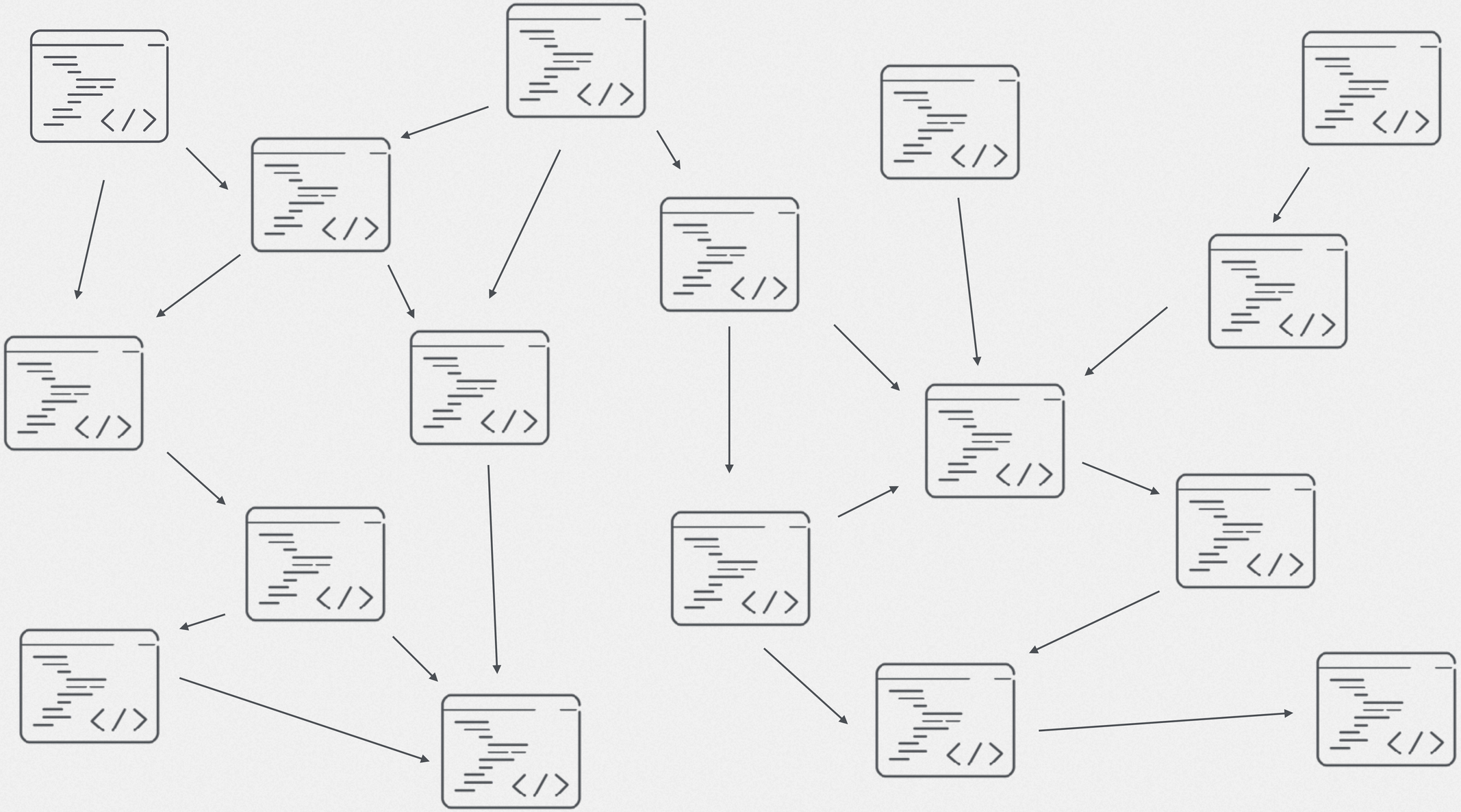
2 **HOW** to even type

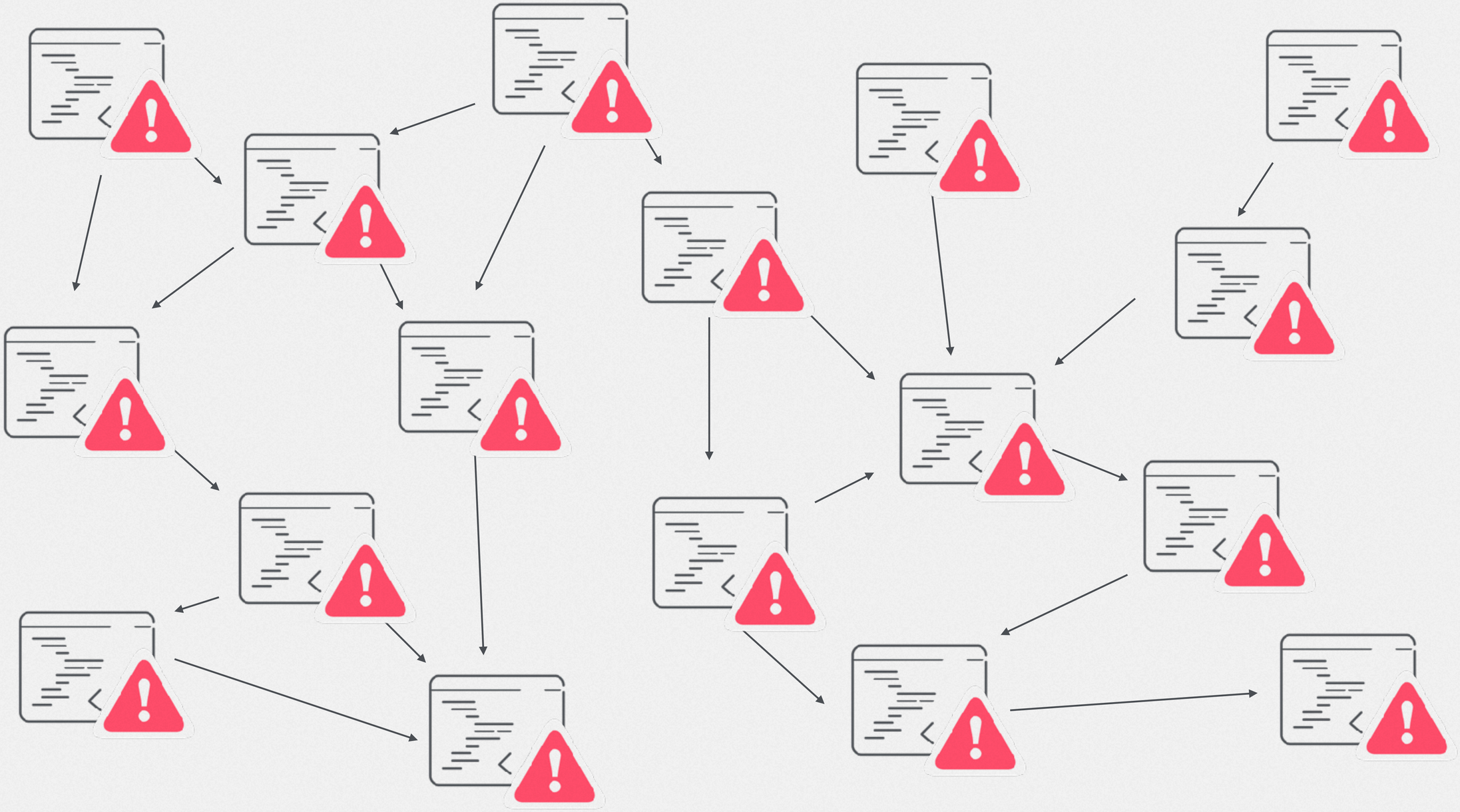
3 **GRADUAL** typing

4 **ISSUES** you may run into



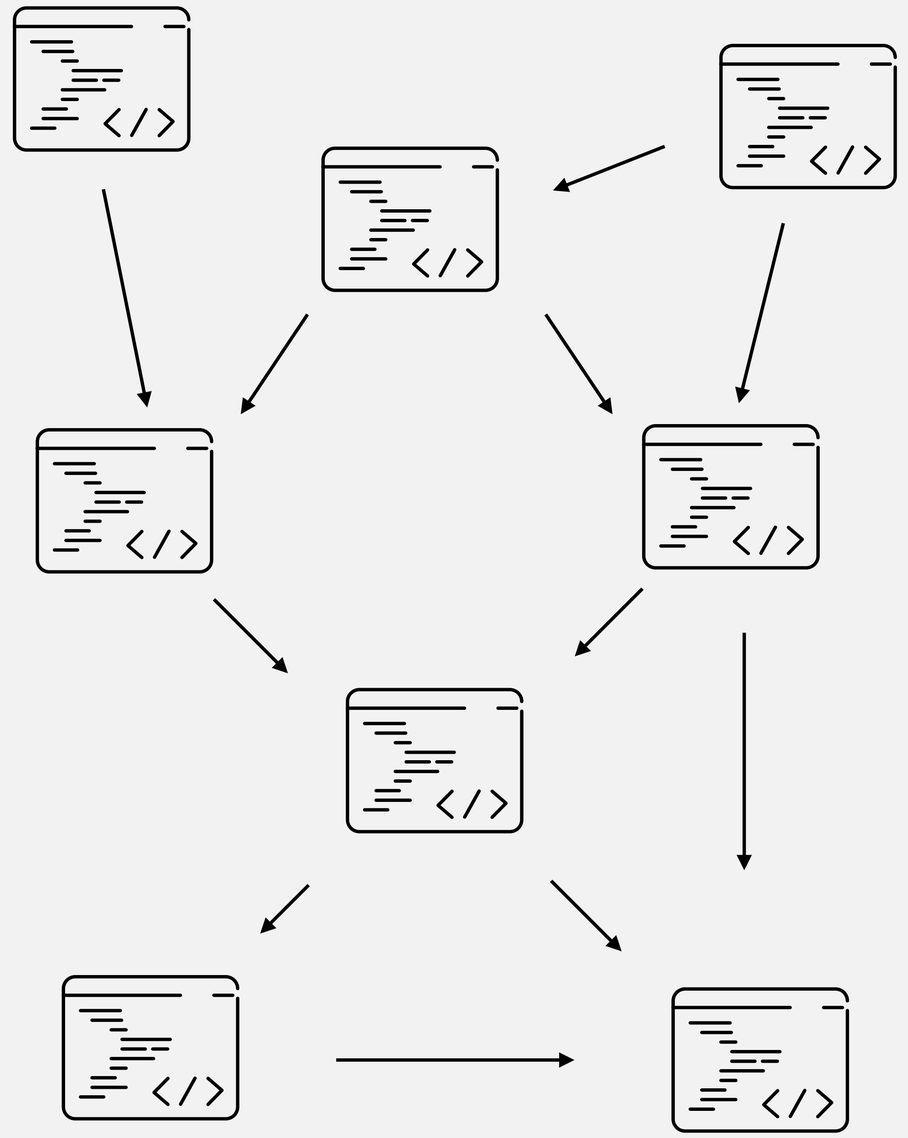
GRADUAL typing





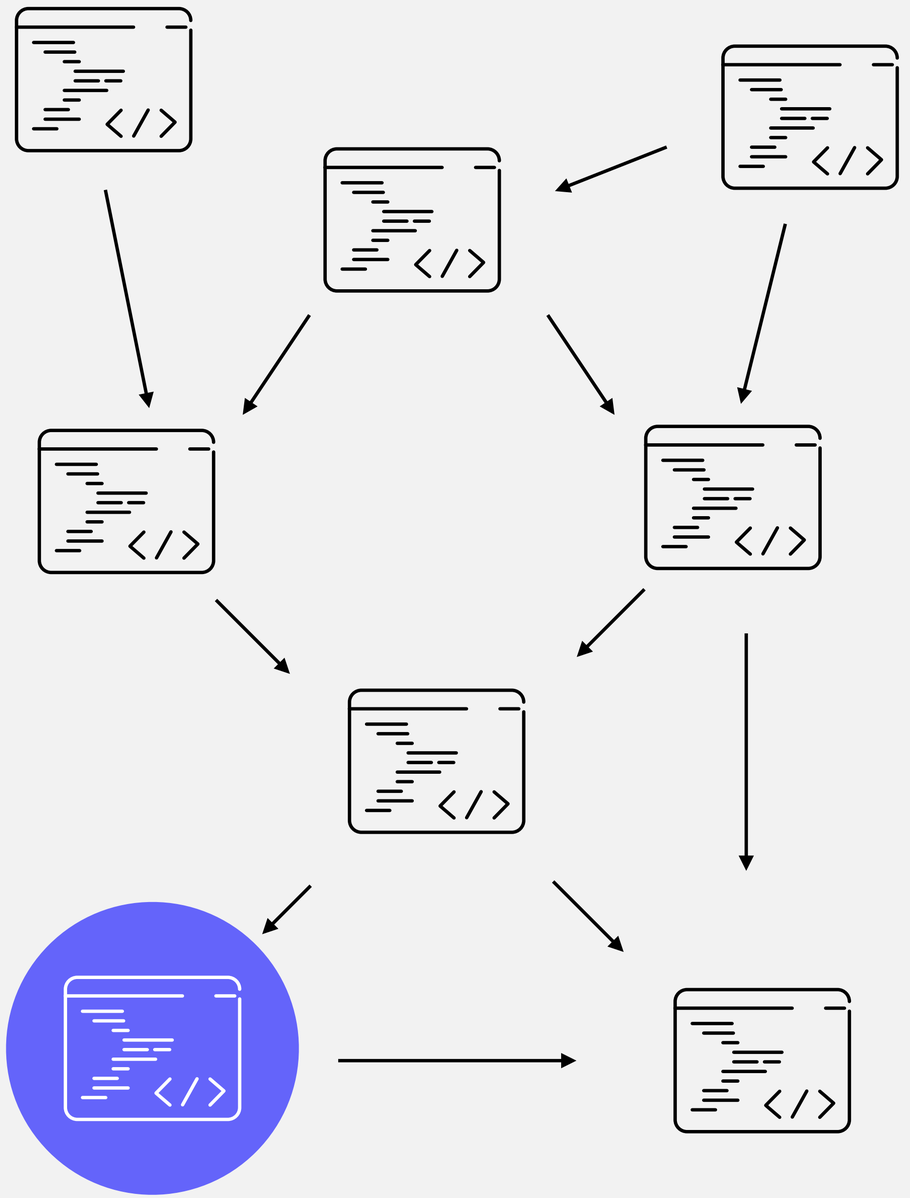
Gradual Typing

- ANNOTATED FUNCTIONS ONLY
- NETWORK EFFECT



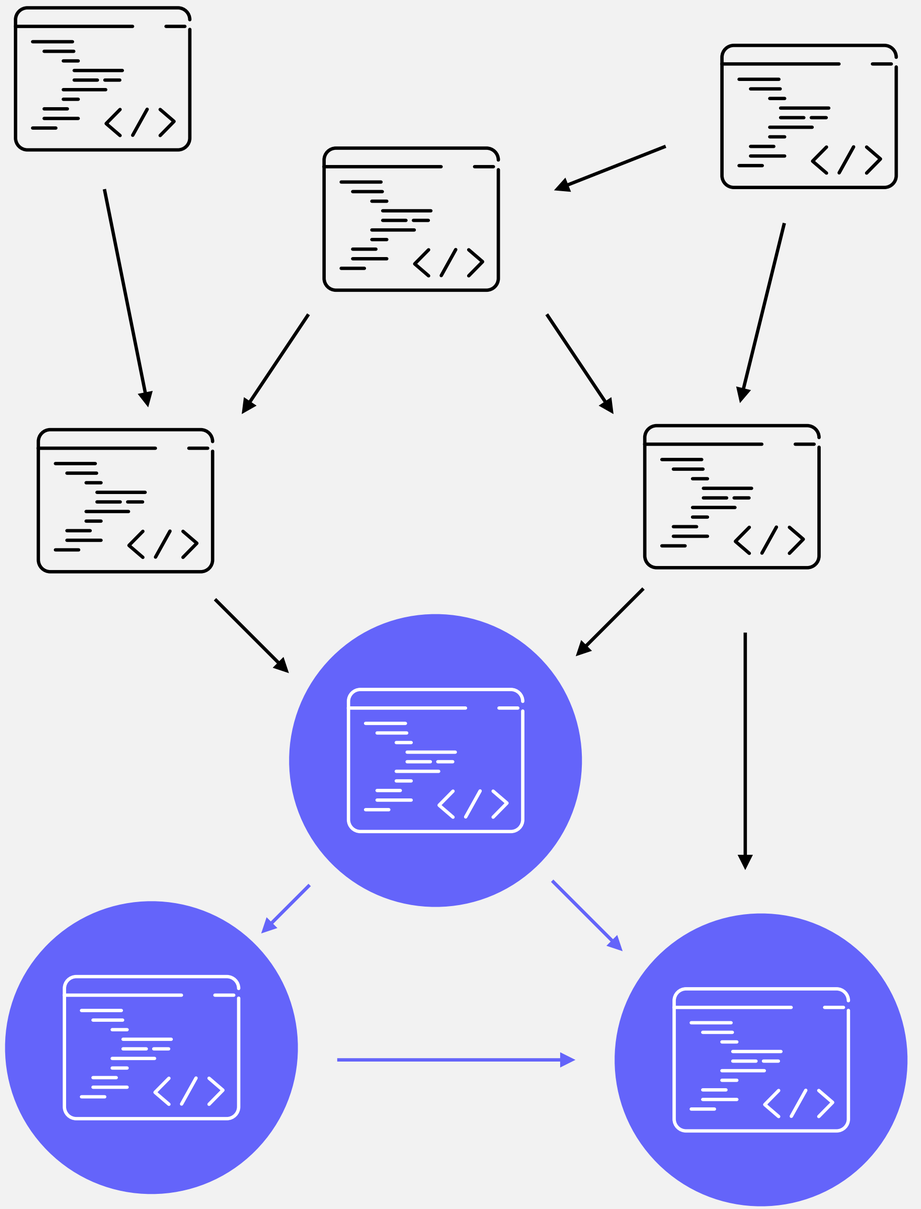
Gradual Typing

- ANNOTATED FUNCTIONS ONLY
- NETWORK EFFECT



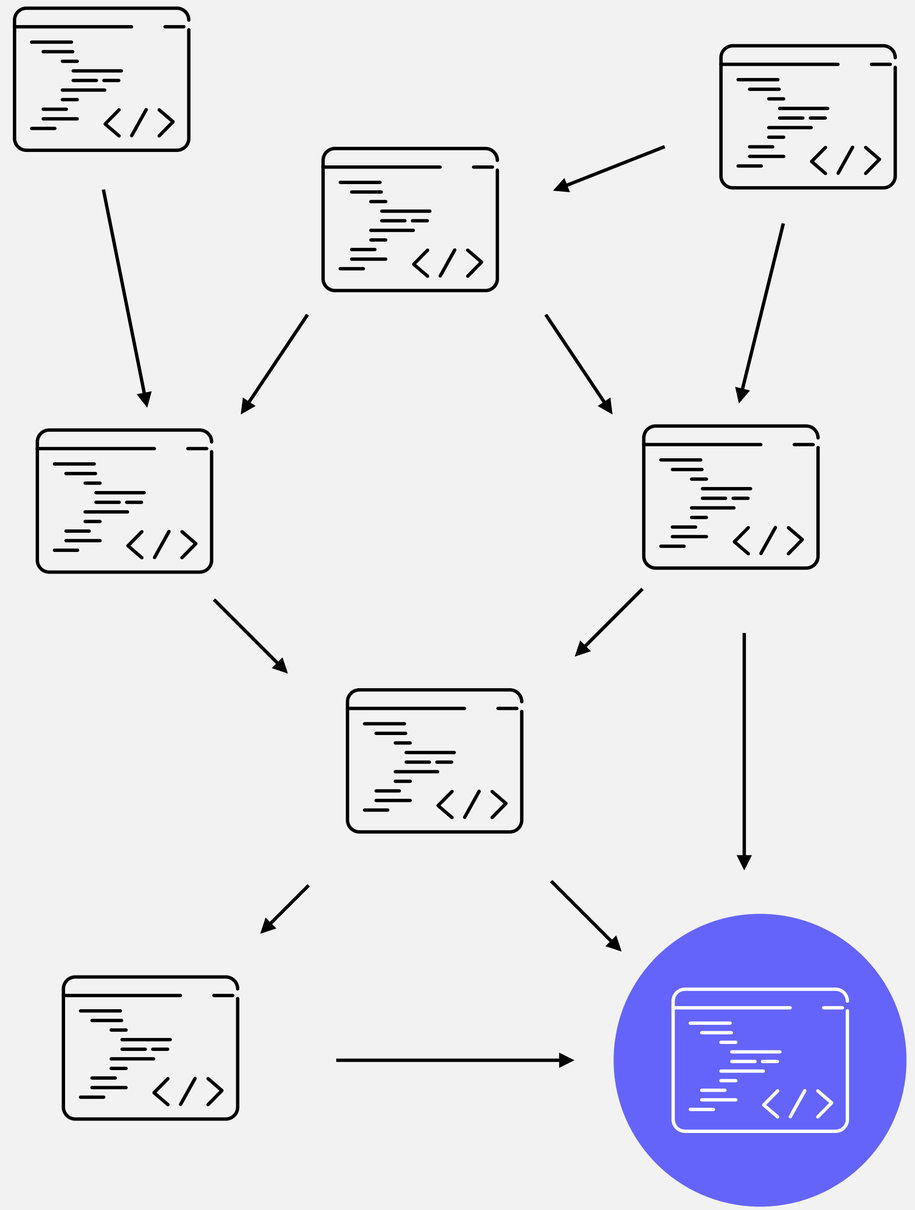
Gradual Typing

- ANNOTATED FUNCTIONS ONLY
- NETWORK EFFECT



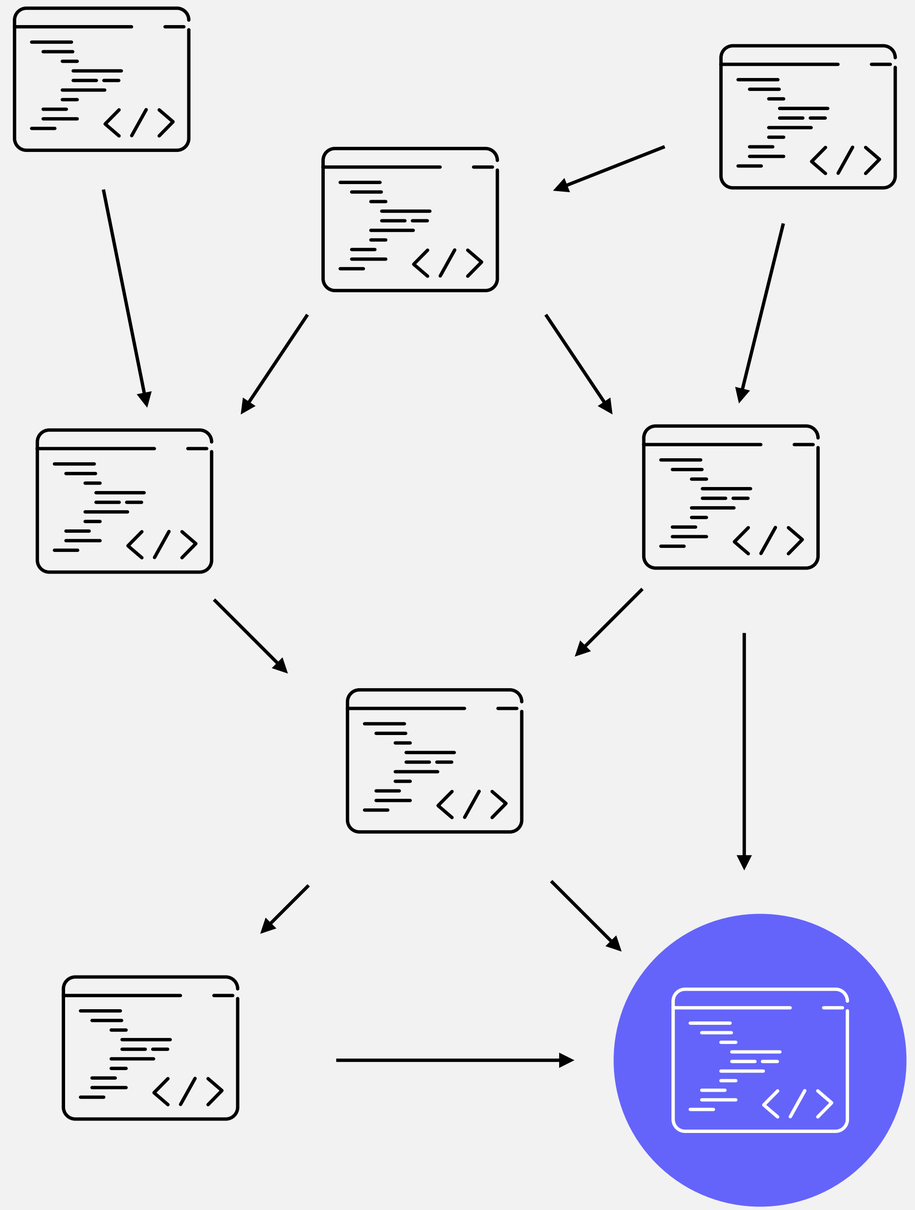
Gradual Typing

- ANNOTATED FUNCTIONS ONLY
- NETWORK EFFECT
- START WITH MOST-USED



Gradual Typing

- ANNOTATED FUNCTIONS ONLY
- NETWORK EFFECT
- START WITH MOST-USED
- USE CI TO DEFEND PROGRESS




MonkeyType

Using Monkeytype

```
$ pip install monkeytype

$ monkeytype run mytests.py

$ monkeytype stub some.module
def myfunc(x: int) -> int: ...

class Duck:
    def __init__(self, name: str) -> None: ...
    def quack(self, volume: int) -> str: ...

$ monkeytype apply some.module
```

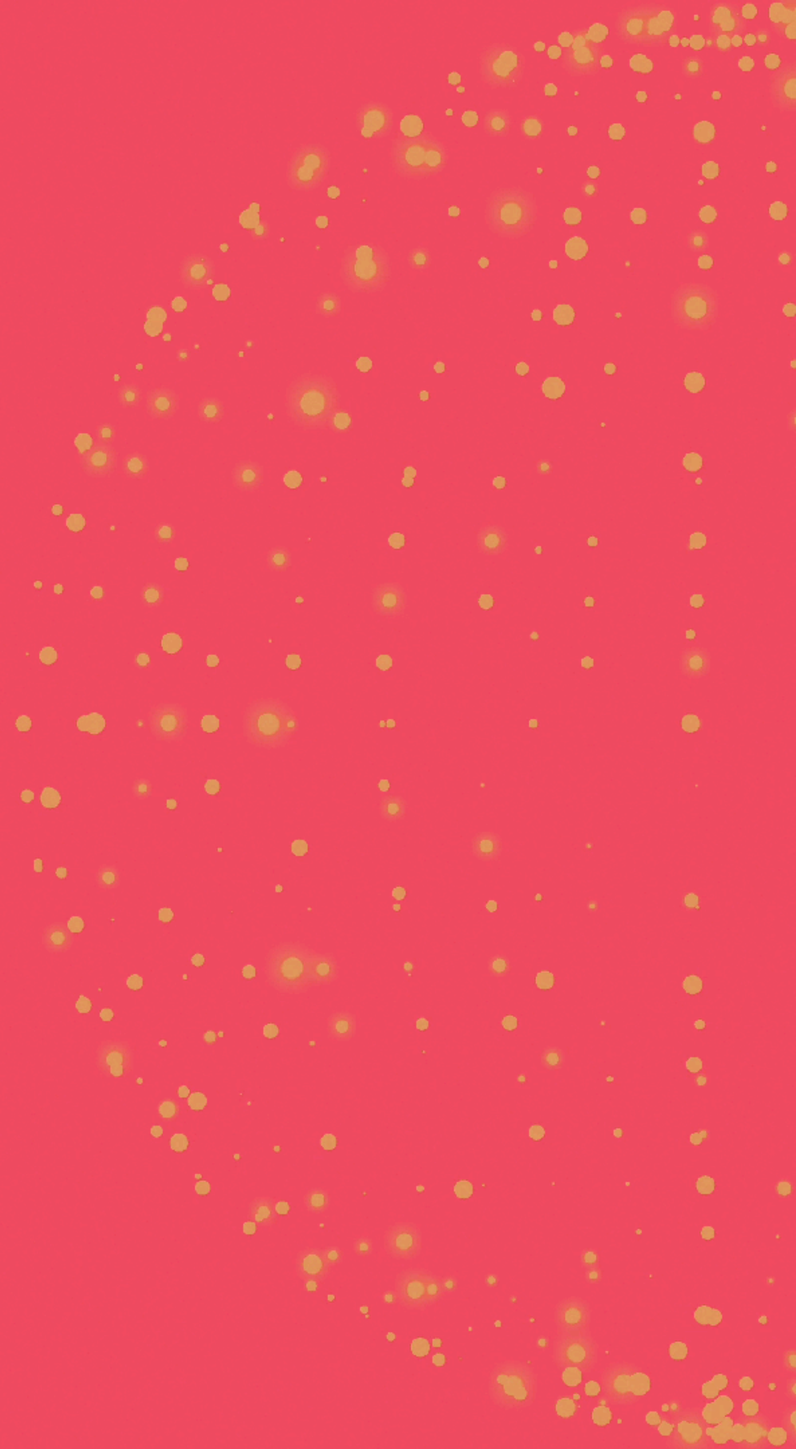
The Plan

1 **WHY** type

2 **HOW** to even type

3 **GRADUAL** typing

4 **ISSUES** you may run into



ISSUES
you may run into

The Apparent Inconsistency

- Why does mypy complain about X here and not there?
- Code in non-annotated functions is not checked!
- Possible fix: visualize type-checked vs not-type-checked?

The Forward Reference


```
class Foo:
    def get_bars(self) -> List[Bar]:
        return get_bars_for_foo(self): ...

class Bar:
    def __init__(self, foo: Foo):
        self.foo = foo
```

The Forward Reference


```
class Foo:
    def get_bars(self) -> List[Bar]:
        return get_bars_for_foo(self): ...

class Bar:
    def __init__(self, foo: Foo):
        self.foo = foo
```

```
NameError: name "Bar" is not defined
```

The Forward Reference


```
class Foo:
    def get_bars(self) -> List['Bar']:
        return get_bars_for_foo(self): ...


class Bar:
    def __init__(self, foo: Foo):
        self.foo = foo
```

The Forward Reference


```
from typing import TYPE_CHECKING

if TYPE_CHECKING:
    from .models import Bar # noqa

def get_bar() -> 'Bar':
    return bar_from_somewhere()
```

The Forward Reference


```
# fix coming in Python 3.7!
from __future__ import annotations

class Foo:
    def get_bars(self) -> List[Bar]:
        return get_bars_for_foo(self)

class Bar:
    def __init__(self, foo: Foo):
        self.foo = foo
```

The Forward Reference


```
# fix coming in Python 3.7!
from __future__ import annotations

class Foo:
    def get_bars(self) -> List[Bar]:
        return get_bars_for_foo(self)


class Bar:
    def __init__(self, foo: Foo):
        self.foo = foo
```

The "Mutable Containers Are Invariant"


```
def process(nums: List[Optional[int]]):
    ...

def foo(nums: List[int]):
    process(nums)
```

The "Mutable Containers Are Invariant"


```
def process(nums: List[Optional[int]]):
    ...

def foo(nums: List[int]):
    process(nums)

# incompatible type "List[int]";
#  expected "List[Optional[int]]"
```

The "Mutable Containers Are Invariant"


```
def process(nums: Sequence[Optional[int]]):
    ...

def foo(nums: List[int]):
    process(nums)
```

The "Mutable Containers Are Invariant"


```
def process(objs: List[Base]):
    ...

def foo(objs: List[SubBase]):
    process(objs)
```

The "Mutable Containers Are Invariant"


```
def process(objs: List[Base]):
    ...

def foo(objs: List[SubBase]):
    process(objs)

# incompatible type "List[SubBase]";
#   expected "List[Base]"
```

The "Mutable Containers Are Invariant"


```
def process(objs: Sequence[Base]):
    ...

def foo(objs: List[Base]):
    process(objs)
```

The Type Shed

- The Python standard library has no type annotations.
- But stdlib annotations would be really useful!
- Instead: **github.com/python/typeshed**
- When mypy says "no such attribute" on a stdlib type, but it clearly exists: typeshed bug!

The Legacy Config Defaults


```
# you _always_ want this option:

--no-implicit-optional

# if you're starting a greenfield project,
# you might want this:

--strict
```

The Future...

Python 3.7: no more ugly string forward references.

Fewer imports from typing module: `dict` not `typing.Dict`

PEP 561: bundling type stubs with third-party packages

Using type annotations to improve performance?

Runtime type enforcement?

Type-checked Python is here and it works!

It catches bugs, and developers like it.

With MonkeyType, you can annotate large legacy codebases.

Early days, far from perfect, but Good Enough.

It'll get better!

THANK
YOU!

MYPY DOCUMENTATION: MYPY.READTHEDOCS.IO

THE REFERENCE STANDARD: PYTHON.ORG/DEV/PEPS/PEP-0484/

REALTIME SUPPORT: GITTER.IM/PYTHON/TYPING

PEP 484 ISSUES: GITHUB.COM/PYTHON/TYPING

TYPE CHECKER ISSUES: GITHUB.COM/PYTHON/MYPY

STDLIB ANNOTATION ISSUES: GITHUB.COM/PYTHON/TYPESHED

MONKEYTYPE ISSUES: GITHUB.COM/INSTAGRAM/MONKEYTYPE

SEE YOU ON THE INTERNETS!

- CARLJM ON IRC (#PYTHON)
- @CARLJM
- CARLJM@INSTAGRAM.COM
- INSTAGR.AM/CARL.J.MEYER