Typed CellProfiler

Incremental type annotations in CellProfiler

Types

A set of values, and a set of functions that can be applied to those values

Defining Types:

- Specify full set of allowable values
    - `Bool: True | False`
- Specify functions which can be used with variables of the type
    - `Sized`: all objects that have a `__len__` method, eg:
        - `[1, 2, 3]`
        - `"abc"`
- Class definition

```python
class UserId(int):
    ...
```

-
    - all instances of `UserId` form a type
- More complex, composite types
    - `String`: a subset of `List` such that all elements are of type `Char`, ie `List[Char]`
    - `Version`: a union of `Integer`, and `String`

Subtypes

A hierarchy of types

Every Type is a Subtype of itself

The set of values of a subtype is a subset of its supertype’s set of values

- **The set of values becomes smaller** in the process of subtyping, or stays the same size

The set of functions of a subtype is a superset of its supertype’s set of functions

- **The set of functions becomes larger** in the process of subtyping, or stays the same size

Subtypes

```
a: Float
b: Integer
```

`` `a = b` ``

- Okay
- Every possible value of `` `b` `` is also in the set of values of `` `a` ``
- Every function from `` `a` `` is also in the set of functions of `` `b` ``
- `` `b` `` (`` `Integer` ``) is a subtype of `` `a` `` (`` `Float` ``)

`` `b = a` ``

- Not okay (unless you're purposely upcasting)
- There are some values of `` `a` `` not in the set of values of `` `b` ``
- There are functions `` `b` `` has that `` `a` `` doesn't (e.g. bitwise shifts)

Subtypes

`List[Int]` is **NOT** a subtype of `List[Float]`

- set of values of `List[Int]` are contained in the set of values of `List[Float]`
- set of functions of `List[Float]` are **NOT** contained in the set of functions of `List[Int]`
  - `append_float()`
- *invariance* relationship

Types vs Classes

`Class` is a dynamic, runtime concept

- classes are object factories
- returned by `type(obj)` builtin

Types are static, and used by static type checkers outside of runtime

Classes define types, but static types not to be confused with runtime classes

- `int` is a class and type
- `UserId` is a class and type
- `Union[str, int]` is a type, but not a class

Python Typing

Brief history

Jukka Lehtosalo - 2010, PhD research in unifying dynamic and statically typed languages

- Gradual growth from untyped prototype to statically typed product
- Converted research to mypy - an optional static type checker for Python
- Introduced at PyCon 2013, gaining interest of Guido van Rossum
- Joined Dropbox, which implemented mypy on a 4 million+ line codebase

Introduced in 2015: Python 3.5, PEP-484

- Type hinting (type annotation)
- Typing module

Incorporates Gradual Typing

- Formal work by Jeremy Siek and Walid Taha in 2006
- Allows parts of a program to be dynamically typed and other parts to be statically typed

Gradual Typing

No runtime implications of types*

- hence type *annotations*

Types never required

- hence type *hints*

Dynamic **and** static typing

- annotate only part of a program

No "overhaul" necessary

Type as you go, when desired, only to aid readability and understanding

Gradual Typing

Consistent with... relationship

Type `T1` is *consistent with* type `T2` if `T1` is a subtype of `T2` (but not the other way around)

Type `Any` is *consistent with* every type (but `Any` is not a subtype of `Any`)

Every type is *consistent with* `Any` (but every type is not a subtype of `Any`)

Therefore, `Any` is the set of all values, and the set of all functions on those values

- Both top and bottom of type hierarchy
- *Dynamic type*
- Most things are `Any` unless explicitly annotated otherwise

Motivation

Python is dynamically typed on purpose

- Great for notebooks, scripts, small scale applications
- Increasingly bad at scale
- Great for single developer projects
- Bad for collaborative projects

Motivation

A type checker will find many subtle (and not so subtle) bugs

- e.g. forgetting to handle a `None` return value

Motivation

Increased readability

- New contributors more easily able to understand what a method does, and how to reuse it
- Refactoring is easier

Motivation

Type checking much faster than testing (for correct types)

Type checkers built into popular IDEs

- Code completion
- Error highlighting
- Go to definition

Primitave Types

Simple types

Start with atomic types: `str`, `int`, `bool`, `float`, `None`, ...

`def is_valid_path(path: str) -> bool:`

Typing Module

Runtime support for type hints

```python
from typing import Any, Union, Optional, Tuple, Callable, Intersection
```

Python Typing

Building blocks

`Any` is consistent with every type, and vice versa

`Optional[t1] == Union[t1, None]`

`Union[t1, t2, ...]`: all types that are subtypes of `t1`, `t2`, ...

- `Union[int, str]` is a subtype of `Union[int, float, str]`
  - *covariance* relationship
- If `ti` is already a `Union`, it is flattened
- Order doesn't matter
- `Union[t1] == t1`

`Tuple[t1, t2, ..., tn]`: tuple where first element is `t1`, second `t2`, ...

- `Tuple[u1, u2, ..., um]` is a subtype IFF `n == m` and `u1` is a subtype of `t1`, `u2`, of `t2`, ...
- variadic homogeneous tuple: `Tuple[t1, ...]` (literal ellipse)

Python Typing

Building blocks

`Callable[[t1, t2, ..., tn], tr]`: function with positional args `t1` etc., and return type `tr`

- argument list unchecked with `Callable[..., tr]` (literal ellipse)
- `Callable[[], int]` is a subtype of `Callable[[], float]`
  - *covariance* relationship in return type
- `Callable[[float], None]` is a subtype of `Callable[[int], None]`
  - *contravariance* relationship in args
  - counterintuitive, but true

Python Typing

Building blocks

`Intersection[t1, t2, ... ]`: Types that are a subtype of each of `t1`, etc.

- Order doesn’t matter
- Nested intersections flattened
- Intersection of few types superset of intersection of more types
  - e.g. `Intersection[int, str]` superset of `Intersection[int, float, str]`
- `Intersection[t1]` is just `t1`
- `Intersection[t1, t2, Any] == Intersection[t1, t2]`

Function Annotation

```python
from typing import List

def reverse(words: List[str], inplace: bool=False) -> List[str]:
  if inplace:
    words.reverse()
        return words
  else:
        return words[::-1]
```

Void Functions, Optional Arguments

```python
from typing import Optional

def write_config(
        key,
        value,
        config_file_path: Optional[str]=None
    ) -> None:
  ...
```

Variable Annotation

```
class User:
  id: int
  username: str

  def register(self, username: str) -> None:
    self.id = self.generate_new_id()
    self.username = username

def get_user_info(user: User):
    user_id: int = user.id
    ...
```

Enforced Constants

```python
from typing import Final

MY_ID: Final[int] = 123

MY_ID = 456 # not okay, type checker will complain
```

Literal values

```
from typing import Literal

primary_color: Literal["red", "blue", "green"] = "red"
primary_color = "green"
primary_color = "PINK" # not okay, type checker will complain
```

Note: probably best just to use an `` `Enum` `` in the above example, but you get the point

Type Defintions and Aliases

```python
from typing import List, Tuple, NewType

Protocol   = NewType('Protocol', str)  # type definition
Domain     = NewType('Domain', str)
Port       = NewType('Port', int)
Host       = Tuple[Protocol, Domain, Port]  # type alias
UserId     = int | str
Creds      = Tuple[UserId, str]
Connection = Tuple[Host, Creds]

def connect(connections: List[Connection]): # List[Tuple[Tuple[str, str, int], Tuple[int|str, str]]
    pass

connect([
    (
        (Protocol("http://"), Domain("hostname.com"), Port(8080)),
        ("123", "password123")
    ),
    (
        (Protocol("ssh://"), Domain("myserver@192.168.1.100:host"), Port(22)),
        (456, "admin")
    )
])
```

Why Define Types?

```
from typing import NewType
UserId = NewType(‘UserId’, int) # user ids are integers

# elsewhere
def generate_dummy_user():
  user_name = “user1”
  user_id = UserId(123)
```

Later...

```
from typing import NewType
UserId = NewType('UserId', str) # ← decide to change user ids to strings

# elsewhere
def generate_dummy_user():
  user_name = "user1"
  user_id = UserId(123) # ← type checker will tell you to change this
```

We can change `` `UserId` `` to be `` `str | int` ``, or we can change all the places that define suer is to now be strings (and the type checker will tell us all the places to do that)

Alternatively, Later...

```
class UserId:
  def __init__(self, id: int):
    self.id = id
  def encode_id(self) -> str:
    ...

# elsewhere
def generate_dummy_user():
    user_name = "user1"
    user_id = UserId(123)
```

`UserId` is still a subtype of `int`, but now has additional behaviour

Defining Types

```
from typing import NewType

UserId = NewType('UserId', int)

combinedId = UserId(123) + UserId(456) # resulting type int
combinedId = UserId( UserId(123) + UserId(456) ) # resulting type UserId
```

Type Alias vs Type Definition

`Alias = Original` is an equivilance relationship

`Derived = NewType('Derived', Original)` specifies that `Derived` is a subtype of `Original`

- `variable: Derived = ...` means `variable` cannot be `int`, it must be `Derived`
- `variable: int = ...` means `variable` can be `int` or `Derived`

Generics

Generic types

Generic type constructor

- Like a function for types
- Takes a type, and "returns" a new type

Generic Functions

Consistency rather than specificity

```python
def add(x, y):
    return x + y

add(1,2) == 3
add('1', '2') == '12'
add(1.1, 2.2) == 3.3
```

`add` has 2 parameters, that may be of type `int`, `float`, or `str`, so long as both are the *same* type

Generic Functions

Annotating generics

```
T = TypeVar('T', int, float, str)

def add(x: T, y: T) -> T:
    return x + y

add(1,2) # ok
add('1', '2') # ok
add(1.1, 2.2) # ok
add(1, 2.2) # ok
add(1, '2') # not ok - fails type check
```

For `` `add(1, 2.2)` ``, `` `1` `` is of type `` `int` ``, which is a subtype of `` `float` ``, so the generic `` `T` `` is calculated to be `` `float` ``

For `` `add(1, '2')` ``, the only type that the two args have in common is `` `object` `` which is not listed in our constraints of `` `int, float, str` ``

Generic Functions

Annotating generics

```python
from typing import Sequence

T = TypeVar('T')

def first(l: Sequence[T]) -> T:
    return l[0]
```

Generic Classes

User-defined generic types

```python
from typing import Generic, TypeVar, Optional

T = TypeVar('T')
S = TypeVar('S', str, int)

class DataStore(Generic[T, S]):

    def put(self, data: T) -> S:
        ...
    def get(self, id: S) -> T:
        ...

ds: DataStore[str, int] = DataStore()
```

Overloading

Functions supporting multiple different combinations of argument types

```
from typing import overload, Optional, List

@overload
def query_datastore(key: None) -> bool: # True if datastore online
    ...

@overload
def query_datastore(key: str) -> Optional[str]: # None if no match
    ...

@overload
def query_datastore(key: List[str]) -> List[str]: # Empty list if none match
    ...

def query_datastore(key):
    pass # actual implementation
```

Series of overloads must be followed by exactly one implementation

Nominative vs Structural Subtyping

PEP-544 introduced *structural* typing as opposed to *nominative* typing

Nominative Subtyping

Subtyping by name

In *nominative* subtyping, `B` is declared to be a subclass of `A`

```
class A:
    def some_method(self) -> int:
        ...

class B(A):
    def some_method(self) -> int:
        return 0

def call_some_method(o: A) -> int:
    return o.some_method()

call_some_method( B() )
```

Structural Subtyping

Subtyping by shape

In *structural* subtyping, `B` looks and acts like `A`, and is therefore a subclass of `A`

- also known as *static duck typing*

```python
class A:
    def some_method(self) -> int:
        ...

class B:
    def some_method(self) -> int:
        return 0

def call_some_method(o: A) -> int:
    return o.some_method()

call_some_method( B() ) # python type checker would complain, B is not A
```

Protocols

Custom protocols through Protocol Class

- terribly named

1. define a class that specifies a set of methods (a "protocol")
2. which defines a type
3. any class with that same set of methods (same "protocol") is considered a subtype
4. without having to declare itself as a subtype, explicitly

Protocols

Structural subtyping enabled

```python
from typing import Protocol

class A(Protocol):
    def some_method(self) -> int:
        ...

class B:
    def some_method(self) -> int:
        return 0

def call_some_method(o: A) -> int:
    return o.some_method()

call_some_method( B() ) # okay, type checker no longer complains
```

Duck Typing

```
def mult_add(x, y, b):
    return x * y + b
```

Type of `x`, `y`, and `b` not important, so long as:

- `x` supports multiplication with `y`, via `__mul__`
- result of `x * y` supports addition with `b`, via `__add__`

```
class Prop:
    def __init__(self, s_val):
        self.s_val = s_val
    def __mul__(self, prop_other):
        return Prop( self.s_val * len(prop_other.s_val) )
    def __add__(self, prop_other):
        return Prop( self.s_val + prop_other.s_val )
    def __repr__(self):
        return self.s_val

mult_add( Prop("foo"), Prop("bar"), Prop("baz") )
# > foofoofoobaz
```

Static Duck Typing

```
from typing import Protocol

class MULT_ADD(Protocol):
    def __mul__(self, other):
        ...
    def __add__(self, other):
        ...

class Prop:
    def __init__(self, s_val):
        self.s_val = s_val
    def __mul__(self, prop_other):
        return Prop( self.s_val * len(prop_other.s_val) )
    def __add__(self, prop_other):
        return Prop( self.s_val + prop_other )

def mult_add(x: MULT_ADD, y: MULT_ADD, b: MULT_ADD):
    return x * y + b

mult_add(Prop("foo"), Prop("bar"), Prop("baz")) # okay

mult_add(2, 3.0, 4.5+6.7j) # also okay
```

Progressive Typing

Initial cleanup

- We already have type errors
  - Mostly related to `None` and `Unknown`
  - Some ignorable: untyped external libraries (e.g. wxPython)
- Should be the only "painful" part (for me)

Add types when desired, to help yourself understand the code

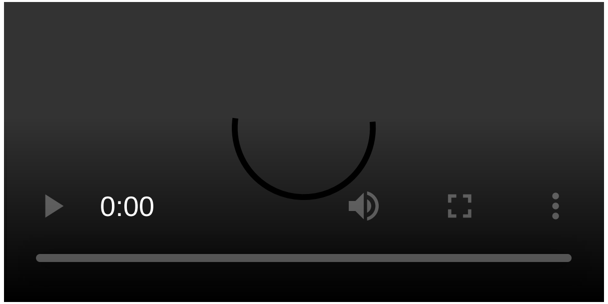
- Goal is not eventually reaching 100% type coverage

Gradually increasing strictness requirements for new code

Eventually add type checking to CI

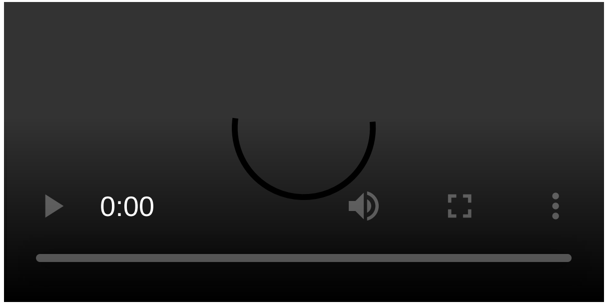
Typed CP

Untyped



Typed CP

Typed



Tools

Type checkers

- mypy
  - mypy playground
- pytype by Google
  - Type annotations generation
- pyre-check by Facebook
- pyright used in vscode
- pylint - static code analysis

Stub files for type definitions

- typeshed repo

Resources

- PEP-3107: Function Annotations
- PEP-483: The Theory of Type Hints
- PEP-484: Type Hints
- PEP-544: Protocols: Structural subtyping (static duck typing)
- Other relevant PEPs
- Dropbox - our journey to type checking 4 million lines of Python
- Gradual Typing
- How to enable Python type checking in VSCode