Navigating the Python Type Jungle

Andrei Nacu, Dorel Lucanu

Faculty of Computer Science, UAIC Iasi

FROM 2025 (Working Formal Methods Symposium)

Table of Contents

1. Introduction
   - Context
   - Type Annotations
   - Duck Typing
   - Everything is an Object
   - Classes and Metaclasses
   - ABCs and Protocols
2. Static Type System Proposal
   - Existential Types
   - Static Type System for Python
     - The Foundational (Blueprint) Layer
     - The Meta Layer
3. Conclusion and Future Work

Table of Contents

1. Introduction
   - Context
   - Type Annotations
   - Duck Typing
   - Everything is an Object
   - Classes and Metaclasses
   - ABCs and Protocols
2. Static Type System Proposal
   - Existential Types
   - Static Type System for Python
     - The Foundational (Blueprint) Layer
     - The Meta Layer
3. Conclusion and Future Work

Introduction

- Python, a very popular programming language (TIOBE, PYPL)

Introduction

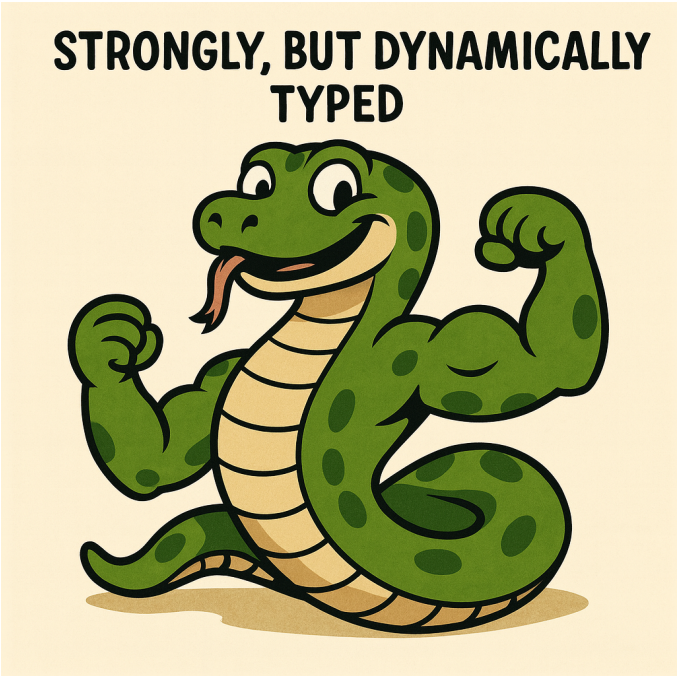
- Python, a very popular programming language (TIOBE, PYPL)
- Pragmatic evolution $\Rightarrow$ flexible and powerful system

Introduction

- Python, a very popular programming language (TIOBE, PYPL)
- Pragmatic evolution $\Rightarrow$ flexible and powerful system
- Scattered documentation: PEPs, module documentation, etc.

Introduction

- Python, a very popular programming language (TIOBE, PYPL)
- Pragmatic evolution $\Rightarrow$ flexible and powerful system
- Scattered documentation: PEPs, module documentation, etc.
- Strongly, but dynamically typed programming language



Type Annotations

Type Annotations

- Type annotations allow the declaration of the expected type of variables, function parameters and return values.

```
def add(x: int, y: int)
    -> int:
    return x + y
z: int = add(2, 3)   # 5
```

Type Annotations

Type Annotations

- Type annotations allow the declaration of the expected type of variables, function parameters and return values.
- Type annotations are not enforced at runtime.

```
def add(x: dict, y: dict
    ) -> list:
    return x + y
z: set = add(2, 3)  # 5
```

Type Annotations

Type Annotations

- Type annotations allow the declaration of the expected type of variables, function parameters and return values.
- Type annotations are not enforced at runtime.
- Used by static type checkers to detect type errors ahead of time.
- Used by IDEs to provide better autocompletion and aid code refactoring.

```
def add(x: dict, y: dict
    ) -> list:
    return x + y
z: set = add(2, 3)  # 5
```

Duck Typing

- *If it walks like a duck and it quacks like a duck, then it must be a duck*.
- The suitability of an object for a given operation is determined by whether it provides the required methods/attributes.
- Actual inheritance is irrelevant as long as the object supports the operations in use.

Duck Typing

```
class Duck:
    def quack(self) -> str:
        return "Quack!"

class Donald:
    def quack(self) -> str:
        return "Nobody knows more about
            quacking than me!"

def do_quack(x}) -> None:
    print (x.quack())

duck = Duck()
do_quack(duck)  # OK
donald = Donald()
do_quack(donald)  # OK
do_quack(3)  # AttributeError
```

Duck Typing

```
class Duck:
    def quack(self) -> str:
        return "Quack!"

class Donald:
    def quack(self) -> str:
        return "Nobody knows more about
            quacking than me!"

def do_quack(x}) -> None:
    print (x.quack())

duck = Duck()
do_quack(duck)   # OK
donald = Donald()
do_quack(donald)   # OK
do_quack(3)   # AttributeError
```

Everything is an Object

In Python, *everything is an object* :

Everything is an Object

In Python, *everything is an object* :

- values

```
print(type(2))
    # <class 'int'>
print((2).bit_length())
    # 2
```

Everything is an Object

In Python, *everything is an object* :

- values
- functions

```
print(type(2))
    # <class 'int'>
print((2).bit_length())
    # 2
```

```
def foo(): pass
print(type(foo))
    # <class 'function'>
```

Everything is an Object

In Python, *everything is an object* :

- values
- functions
- classes

```
print(type(2))
    # <class 'int'>
print((2).bit_length())
    # 2
```

```
def foo(): pass
print(type(foo))
    # <class 'function'>
```

```
class bar: pass
print(type(bar))
    # <class 'type'>
```

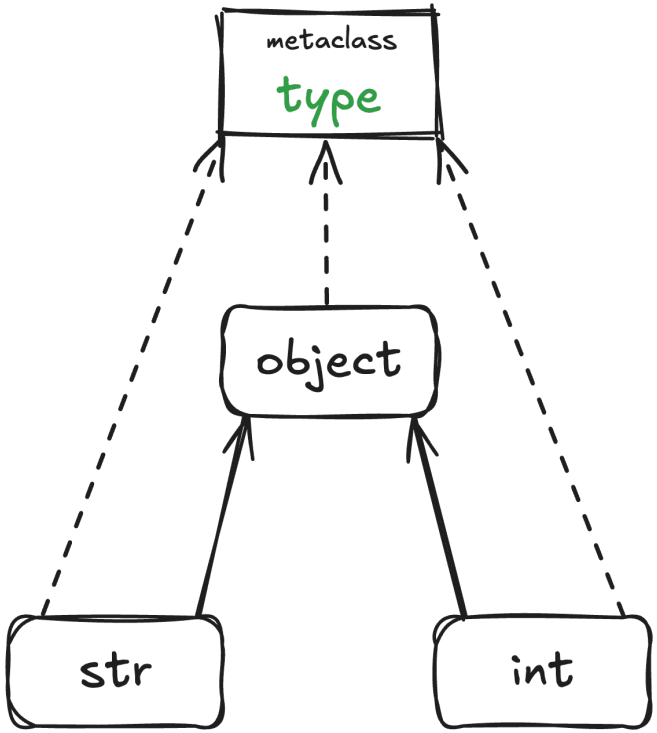
Classes and Metaclasses

- Every Python class is a runtime value as well. It is a *first-class citizen*.

```
class Foo: pass
bar = []
bar.append(Foo)
```

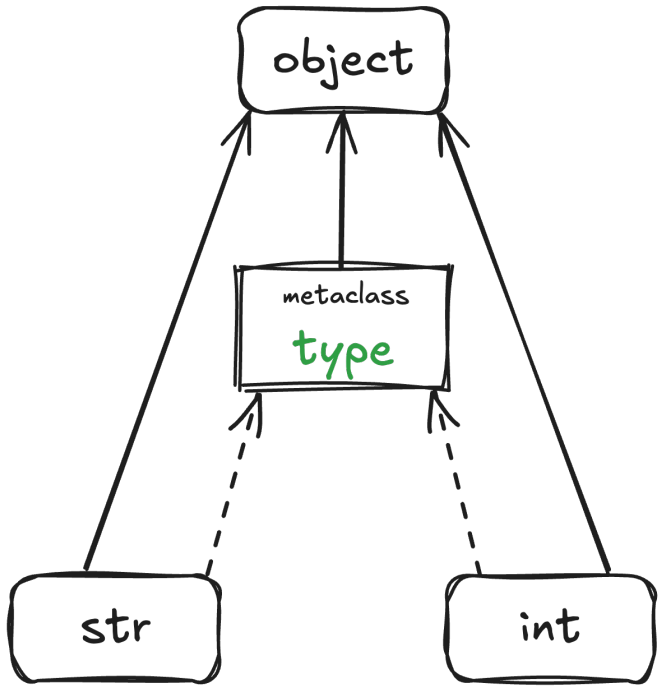
Classes and Metaclasses

- Every Python class is a runtime value as well. It is a *first-class citizen*.
- Every Python class is an instance of the *metaclass* `type`.



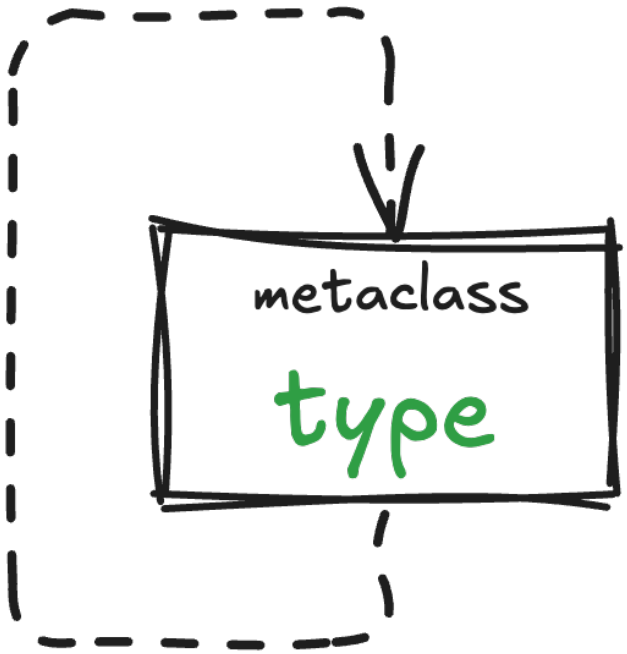
Classes and Metaclasses

- Every Python class is a runtime value as well. It is a *first-class citizen*.
- Every Python class is an instance of the *metaclass* `type`.
- Every Python class is a subclass of the class `object`.



Classes and Metaclasses

- Every Python class is a runtime value as well. It is a *first-class citizen*.
- Every Python class is an instance of the *metaclass* `type`.
- Every Python class is a subclass of the class `object`.
- The *metaclass* `type` is an instance of itself.



ABCs and Protocols

Abstract Base Classes (ABCs)

- **Markers:** classes added into a class inheritance tree to signal that it supports a certain interface.
- ABCs may define a mechanism that allows *virtual subclassing* through `__subclasshook__` and `register`.

ABCs and Protocols

Abstract Base Classes (ABCs)

- **Markers:** classes added into a class inheritance tree to signal that it supports a certain interface.
- ABCs may define a mechanism that allows *virtual subclassing* through `__subclasshook__` and `register`.

- **Contracts:** ABCs define a minimal set of methods that subclasses must implement.
- ABCs can also provide concrete methods which provide subclasses a ready-made functionality.

ABCs and Protocols

Abstract Base Classes (ABCs)

- **Markers:** classes added into a class inheritance tree to signal that it supports a certain interface.
- ABCs may define a mechanism that allows *virtual subclassing* through `__subclasshook__` and `register`.

- **Contracts:** ABCs define a minimal set of methods that subclasses must implement.
- ABCs can also provide concrete methods which provide subclasses a ready-made functionality.

- Quirk: Conceptually, abstract classes cannot be instantiated. However, a class that inherits from `abc.ABC` *can* be instantiated if it contains no methods marked as abstract.

ABCs and Protocols

Protocols

[1] https://typing.python.org/en/latest/spec/protocol.html

ABCs and Protocols

Protocols

- **Contracts:** Protocols define a minimal set of methods and attributes that a class must implement to *conform* to it.

[1] https://typing.python.org/en/latest/spec/protocol.html

ABCs and Protocols

Protocols

- **Contracts:** Protocols define a minimal set of methods and attributes that a class must implement to *conform* to it.
- A Protocol cannot be instantiated. *There are no values whose runtime type is a protocol*[1].

[1] `https://typing.python.org/en/latest/spec/protocol.html`

ABCs and Protocols

Protocols

- **Contracts:** Protocols define a minimal set of methods and attributes that a class must implement to *conform* to it.

- A Protocol cannot be instantiated. *There are no values whose runtime type is a protocol*[1].
- Protocols are, esentially, type hinting interfaces. They are used by static type checkers to check type conformity.

[1] `https://typing.python.org/en/latest/spec/protocol.html`

ABCs and Protocols

```
@runtime_checkable  # required for issubclass and isinstance
class MyProtocol(Protocol):
    def f1(self) -> str: ...
    def f2(self) -> int: ...

class Foo:
    def f1(self) -> str:
        return "hello!"
    def f2(self) -> int:
        return 10

class Bar:
    def f1(self) -> str:
        return "hello!"

x = Foo()
y = Bar()
print(issubclass(Foo, MyProtocol))  # True (shallow)
print(isinstance(x, MyProtocol))  # True (shallow)
print(issubclass(Bar, MyProtocol))  # False
print(isinstance(y, MyProtocol))  # False
```

ABCs and Protocols

```
@runtime_checkable  # required for issubclass and isinstance
class MyProtocol(Protocol):
    def f1(self) -> str: ...
    def f2(self) -> int: ...

class Foo:
    def f1(self) -> str:
        return "hello!"
    def f2(self) -> int:
        return 10

class Bar:
    def f1(self) -> str:
        return "hello!"

x = Foo()
y = Bar()
print(issubclass(Foo, MyProtocol))  # True (shallow)
print(isinstance(x, MyProtocol))  # True (shallow)
print(issubclass(Bar, MyProtocol))  # False
print(isinstance(y, MyProtocol))  # False
```

Table of Contents

1. Introduction
   - Context
   - Type Annotations
   - Duck Typing
   - Everything is an Object
   - Classes and Metaclasses
   - ABCs and Protocols
2. Static Type System Proposal
   - Existential Types
   - Static Type System for Python
     - The Foundational (Blueprint) Layer
     - The Meta Layer
3. Conclusion and Future Work

Existential Types

- Every Python type is represented by a class.

Existential Types

- Every Python type is represented by a class.
- A class is an implementation of an abstract data type (ADT).

Existential Types

- Every Python type is represented by a class.
- A class is an implementation of an abstract data type (ADT).
- An ADT has an existential type.

Existential Types

Existential types. $\exists X.\tau$ - there exists a type X such that τ holds.

- $\exists X$ - some concrete data type X chosen as the representation type.
- τ - describes the signature of the ADT's operations.

Existential Types

Existential types. $\exists X.\tau$ - there exists a type X such that τ holds.

- $\exists X$ - some concrete data type X chosen as the representation type.
- τ - describes the signature of the ADT's operations.

An element of $\exists X.\tau$ is a pair (S, t), consisting of:

- a concrete type S, which substitutes the abstract type X;
- a term t of type $\tau[S/X]$, which represents the concrete implementation of the type, where every free occurrence of X is substituted by S.

Existential Types

Existential types. $\exists X.\tau$ - there exists a type X such that τ holds.

- $\exists X$ - some concrete data type X chosen as the representation type.
- τ - describes the signature of the ADT's operations.

An element of $\exists X.\tau$ is a pair (S, t), consisting of:

- a concrete type S, which substitutes the abstract type X;
- a term t of type $\tau[S/X]$, which represents the concrete implementation of the type, where every free occurrence of X is substituted by S.

Note: To model inheritance, the bounded quantification $\exists X <: T.\tau$ is used.

Existential Types

Generic Existential Types. $\forall Y_1, \ldots, Y_k.\exists X.\tau$ - if $\exists X.\tau$ is an existential type and τ includes universally quantified type variables $Y_1, \ldots, Y_k$.

- $\forall Y_1, \ldots, Y_k$: means *for all types* $Y_1, \ldots, Y_k$, where Y_i are generic type parameters;
- $\exists X$: translates to *there exists a hidden implementation type* X, whose own structure depends on the Y_i type parameters;
- τ is the public interface whose operations are defined in terms of both the public types Y_i and the hidden type X.

Existential Types

Generic Existential Types. $\forall Y_1, \ldots, Y_k.\exists X.\tau$ - if $\exists X.\tau$ is an existential type and τ includes universally quantified type variables $Y_1, \ldots, Y_k$.

- $\forall Y_1, \ldots, Y_k$: means *for all types* $Y_1, \ldots, Y_k$, where Y_i are generic type parameters;
- $\exists X$: translates to *there exists a hidden implementation type* X, whose own structure depends on the Y_i type parameters;
- τ is the public interface whose operations are defined in terms of both the public types Y_i and the hidden type X.

An element of $\forall Y.\tau'$ is a function that, given a type Z, produces a concrete instance of $\tau'[Z/Y]$.
An element of $\forall Y.\exists X.\tau$ is a function that, for each type Z, produces a pair consisting of a type S and a term t of type $\tau[Z/Y][S/X]$.

Proposed Static Type System

Pythonic Type System (PyTS) - static type system that captures the subset of Python types that can be modeled using *existential types*.

- The signature τ is a record type that maps class member names to Python-specific type expressions.
- Type expressions are built from a set of fundamental constructs derived from Python core classes.
- The fundamental constructs/primitives are also existential types.

The Foundational (Blueprint) Layer

Built-in Atomic Existential Types

The Foundational (Blueprint) Layer

Built-in Atomic Existential Types

- *numeric types*: `BoolET`, `IntET`, `FloatET`, `ComplexET`;
- *scalar sequence types*: `StrET`, `BytesET`;
- `ObjectET`, for the `object` class;
- `BottomET`, which contains no values;
- `NoneTypeET`, corresponding to Python's `NoneType` class.

The Foundational (Blueprint) Layer

Built-in Atomic Existential Types

- *numeric types*: `BoolET`, `IntET`, `FloatET`, `ComplexET`;
- *scalar sequence types*: `StrET`, `BytesET`;
- `ObjectET`, for the `object` class;
- `BottomET`, which contains no values;
- `NoneTypeET`, corresponding to Python's `NoneType` class.

Built-in Generic Container Existential Types

The Foundational (Blueprint) Layer

Built-in Atomic Existential Types

- *numeric types*: `BoolET`, `IntET`, `FloatET`, `ComplexET`;
- *scalar sequence types*: `StrET`, `BytesET`;
- `ObjectET`, for the `object` class;
- `BottomET`, which contains no values;
- `NoneTypeET`, corresponding to Python's `NoneType` class.

Built-in Generic Container Existential Types

- *sequence types*: `ListET`, `TupleET`, `BytearrayET`;
- *set types*: `SetET`, `FrozensetET`;
- *mapping type*: `DictET`.

The Foundational (Blueprint) Layer

Signature (τ) type expressions are constructed from:

- Built-in existential types: `IntET`, `StrET`, `ListET[IntET]`, etc.

The Foundational (Blueprint) Layer

Signature (τ) type expressions are constructed from:

- Built-in existential types: `IntET`, `StrET`, `ListET[IntET]`, etc.
- Product types: `IntET` $\times$ `StrET`, `ListET[IntET]` $\times$ `FloatET`, etc.

The Foundational (Blueprint) Layer

Signature (τ) type expressions are constructed from:

- Built-in existential types: `IntET`, `StrET`, `ListET[IntET]`, etc.
- Product types: `IntET` $\times$ `StrET`, `ListET[IntET]` $\times$ `FloatET`, etc.
- Sum types: `IntET` $+$ `StrET`, `ListET[IntET]` $+$ `ListET[FloatET]`, etc.

The Foundational (Blueprint) Layer

Signature (τ) type expressions are constructed from:

- Built-in existential types: `IntET`, `StrET`, `ListET[IntET]`, etc.
- Product types: $\texttt{IntET} \times \texttt{StrET}$, $\texttt{ListET[IntET]} \times \texttt{FloatET}$, etc.
- Sum types: $\texttt{IntET} + \texttt{StrET}$, $\texttt{ListET[IntET]} + \texttt{ListET[FloatET]}$, etc.
- Function types: $\texttt{IntET} \times \texttt{StrET} \rightarrow \texttt{BoolET}$, etc.

The Foundational (Blueprint) Layer

```
class Duck:
    def quack(self) -> str:
        return "Quack!"

class Donald:
    def quack(self) -> str:
        return "Nobody knows more about quacking than me!"
```

The Foundational (Blueprint) Layer

```
class Duck:
    def quack(self) -> str:
        return "Quack!"

class Donald:
    def quack(self) -> str:
        return "Nobody knows more about quacking than me!"
```

Duck and Donald are concrete representations of the existential type:

$$\texttt{QuackET} = \exists Q.\{\texttt{quack} : Q \to \texttt{StrET}\}$$

The Foundational (Blueprint) Layer

```
class Duck:
    def quack(self) -> str:
        return "Quack!"

class Donald:
    def quack(self) -> str:
        return "Nobody knows more about quacking than me!"
```

Duck and Donald are concrete representations of the existential type:

$$\texttt{QuackET} = \exists Q.\{\texttt{quack} : Q \to \texttt{StrET}\}$$

An element of QuackET is a pair (S, t), where:

- Representation type $S = \texttt{Duck}$.
- The term t has the type $\tau[\texttt{Duck}/Q] = \{\texttt{quack} : \texttt{Duck} \to \texttt{StrET}\}$.
- An element of QuackET is the pair $(\texttt{Duck}, \{\texttt{quack} := \texttt{Duck.quack}\})$.

The Foundational (Blueprint) Layer

```
class int:
 def __abs__(self) -> int:
  ...

class float:
 def __abs__(self) -> float:
  ...

class complex:
 def __abs__(self) -> float:
  ...
```

```
class SupportsAbs[T](Protocol):
  def __abs__(self) -> T:
    pass
```

The Foundational (Blueprint) Layer

```
class int:
 def __abs__(self) -> int:
  ...

class float:
 def __abs__(self) -> float:
  ...

class complex:
 def __abs__(self) -> float:
  ...
```

```
class SupportsAbs[T](Protocol):
  def __abs__(self) -> T:
    pass
```

The `SupportsAbs` protocol is a representation of the generic existential type:

$$\texttt{SupportsAbsET} = \forall T.\exists SA.\{\texttt{\_\_abs\_\_} : SA \to T\}$$

The Foundational (Blueprint) Layer

```
class int:
 def __abs__(self) -> int:
  ...

class float:
 def __abs__(self) -> float:
  ...

class complex:
 def __abs__(self) -> float:
  ...
```

```
class SupportsAbs[T](Protocol):
  def __abs__(self) -> T:
    pass
```

The `SupportsAbs` protocol is a representation of the generic existential type:

$$\texttt{SupportsAbsET} = \forall T.\exists SA.\{\texttt{\_\_abs\_\_} : SA \to T\}$$

The `int` class is a concrete representation of the existential type:

$$\texttt{SupportsAbsET[IntET]} = \exists SA.\{\texttt{\_\_abs\_\_} : SA \to \texttt{IntET}\}$$

The Foundational (Blueprint) Layer

```
class int:
 def __abs__(self) -> int:
  ...

class float:
 def __abs__(self) -> float:
  ...

class complex:
 def __abs__(self) -> float:
  ...
```

```
class SupportsAbs[T](Protocol):
  def __abs__(self) -> T:
    pass
```

The `SupportsAbs` protocol is a representation of the generic existential type:

$$\texttt{SupportsAbsET} = \forall T.\exists SA.\{\texttt{\_\_abs\_\_} : SA \to T\}$$

The `int` class is a concrete representation of the existential type:

$$\texttt{SupportsAbsET[IntET]} = \exists SA.\{\texttt{\_\_abs\_\_} : SA \to \texttt{IntET}\}$$

The `float` and `complex` classes are a concrete representations of:

$$\texttt{SupportsAbsET[FloatET]} = \exists SA.\{\texttt{\_\_abs\_\_} : SA \to \texttt{FloatET}\}$$

The Foundational (Blueprint) Layer

The `object` class is the parent class of all Python classes.

The Foundational (Blueprint) Layer

The `object` class is the parent class of all Python classes.

$$
\begin{aligned}
&\texttt{ObjectET} = \exists O.\{ \\
&\quad \texttt{\_\_new\_\_} : \texttt{TupleET}[\texttt{ObjectET}, \ldots] \times \texttt{DictET}[\texttt{StrET}, \texttt{ObjectET}] \to O, \\
&\quad \texttt{\_\_init\_\_} : O \times \texttt{TupleET}[\texttt{ObjectET}, \ldots] \times \texttt{DictET}[\texttt{StrET}, \texttt{ObjectET}] \to O, \\
&\quad \ldots\}
\end{aligned}
$$

The Foundational (Blueprint) Layer

The `object` class is the parent class of all Python classes.

$$\texttt{ObjectET} = \exists O.\{$$
$$\texttt{\_\_new\_\_} : \texttt{TupleET}[\texttt{ObjectET}, \ldots] \times \texttt{DictET}[\texttt{StrET}, \texttt{ObjectET}] \to O,$$
$$\texttt{\_\_init\_\_} : O \times \texttt{TupleET}[\texttt{ObjectET}, \ldots] \times \texttt{DictET}[\texttt{StrET}, \texttt{ObjectET}] \to O,$$
$$\ldots\}$$

- allocate space for the new instance

The Foundational (Blueprint) Layer

The `object` class is the parent class of all Python classes.

$$\texttt{ObjectET} = \exists O.\{$$
$$\texttt{\_\_new\_\_}: \texttt{TupleET}[\texttt{ObjectET}, \ldots] \times \texttt{DictET}[\texttt{StrET}, \texttt{ObjectET}] \to O,$$
$$\texttt{\_\_init\_\_}: O \times \texttt{TupleET}[\texttt{ObjectET}, \ldots] \times \texttt{DictET}[\texttt{StrET}, \texttt{ObjectET}] \to O,$$
$$\ldots\}$$

- allocate space for the new instance
- initialize the new instance

The Foundational (Blueprint) Layer

The `object` class is the parent class of all Python classes.

$$\texttt{ObjectET} = \exists O.\{$$
$$\texttt{\_\_new\_\_}: \texttt{TupleET}[\texttt{ObjectET}, \ldots] \times \texttt{DictET}[\texttt{StrET}, \texttt{ObjectET}] \to O,$$
$$\texttt{\_\_init\_\_}: O \times \texttt{TupleET}[\texttt{ObjectET}, \ldots] \times \texttt{DictET}[\texttt{StrET}, \texttt{ObjectET}] \to O,$$
$$\ldots\}$$

- allocate space for the new instance
- initialize the new instance
- `*args` (tuple with variable number of elements) and `**kwargs`

The Foundational (Blueprint) Layer

The `object` class is the parent class of all Python classes.

$$\texttt{ObjectET} = \exists O.\{$$
$$\texttt{\_\_new\_\_}: \texttt{TupleET}[\texttt{ObjectET}, \ldots] \times \texttt{DictET}[\texttt{StrET}, \texttt{ObjectET}] \to O,$$
$$\texttt{\_\_init\_\_}: O \times \texttt{TupleET}[\texttt{ObjectET}, \ldots] \times \texttt{DictET}[\texttt{StrET}, \texttt{ObjectET}] \to O,$$
$$\ldots\}$$

- allocate space for the new instance
- initialize the new instance
- `*args` (tuple with variable number of elements) and `**kwargs`
- more methods

The Foundational (Blueprint) Layer

- `TypeVar` class existential type:

$$\texttt{TypeVarET} = \exists TV <: \texttt{ObjectET}.\{\texttt{\_\_name\_\_} : \texttt{NoneTypeET} \rightarrow \texttt{StrET}, \ldots\}$$

- `typing.Generic` existential type:

$$\texttt{GenericET}_\texttt{n} = \forall T_1, \ldots, T_n.\exists G <: \texttt{ObjectET}.\{$$
$$\texttt{\_\_parameters\_\_} : \texttt{NoneTypeET} \rightarrow \texttt{TupleET}[\texttt{TypeVarET}, \ldots], \ldots\}$$

The Foundational (Blueprint) Layer

- `TypeVar` class existential type:

$$\texttt{TypeVarET} = \exists TV <: \texttt{ObjectET}.\{\texttt{\_\_name\_\_} : \texttt{NoneTypeET} \rightarrow \texttt{StrET}, \ldots\}$$

- `typing.Generic` existential type:

$$\texttt{GenericET}_\texttt{n} = \forall T_1, \ldots, T_n.\exists G <: \texttt{ObjectET}.\{$$
$$\texttt{\_\_parameters\_\_} : \texttt{NoneTypeET} \rightarrow \texttt{TupleET}[\texttt{TypeVarET}, \ldots], \ldots\}$$

The Foundational (Blueprint) Layer

- `TypeVar` class existential type:

$$\texttt{TypeVarET} = \exists TV <: \texttt{ObjectET}.\{\texttt{\_\_name\_\_} : \texttt{NoneTypeET} \to \texttt{StrET}, \ldots\}$$

- `typing.Generic` existential type:

$$\texttt{GenericET}_\texttt{n} = \forall T_1, \ldots, T_n.\exists G <: \texttt{ObjectET}.\{ \\ \texttt{\_\_parameters\_\_} : \texttt{NoneTypeET} \to \texttt{TupleET}[\texttt{TypeVarET}, \ldots], \ldots\}$$

- `typing.Protocol` existential type:

$$\texttt{ProtocolET}_\texttt{n} = \forall T_1, \ldots, T_n.\exists P <: \texttt{GenericET}_{\texttt{n}+1}[P, T_1, \ldots, T_n].\{ \\ \texttt{\_\_new\_\_} : \texttt{TupleET}[\texttt{ObjectET}, \ldots] \times \texttt{DictET}[\texttt{StrET}, \texttt{ObjectET}] \to \texttt{BottomET}, \\ \texttt{\_is\_protocol} : \texttt{NoneTypeET} \to \texttt{BoolET}, \\ \texttt{\_is\_runtime\_protocol} : \texttt{NoneTypeET} \to \texttt{Bool}, \ldots\}$$

The Foundational (Blueprint) Layer

- `TypeVar` class existential type:

$$\texttt{TypeVarET} = \exists TV <: \texttt{ObjectET}.\{\texttt{\_\_name\_\_} : \texttt{NoneTypeET} \to \texttt{StrET}, \ldots\}$$

- `typing.Generic` existential type:

$$\texttt{GenericET}_\texttt{n} = \forall T_1, \ldots, T_n.\exists G <: \texttt{ObjectET}.\{ \texttt{\_\_parameters\_\_} : \texttt{NoneTypeET} \to \texttt{TupleET}[\texttt{TypeVarET}, \ldots], \ldots\}$$

- `typing.Protocol` existential type:

$$\begin{aligned}\texttt{ProtocolET}_\texttt{n} = {} & \forall T_1, \ldots, T_n.\exists P <: \texttt{GenericET}_{\texttt{n}+1}[P, T_1, \ldots, T_n].\{ \\ & \texttt{\_\_new\_\_} : \texttt{TupleET}[\texttt{ObjectET}, \ldots] \times \texttt{DictET}[\texttt{StrET}, \texttt{ObjectET}] \to \texttt{BottomET}, \\ & \texttt{\_is\_protocol} : \texttt{NoneTypeET} \to \texttt{BoolET}, \\ & \texttt{\_is\_runtime\_protocol} : \texttt{NoneTypeET} \to \texttt{Bool}, \ldots\}\end{aligned}$$

The Foundational (Blueprint) Layer

- `TypeVar` class existential type:

$$\texttt{TypeVarET} = \exists TV <: \texttt{ObjectET}.\{\texttt{\_\_name\_\_} : \texttt{NoneTypeET} \to \texttt{StrET}, \ldots\}$$

- `typing.Generic` existential type:

$$\texttt{GenericET}_\texttt{n} = \forall T_1, \ldots, T_n.\exists G <: \texttt{ObjectET}.\{ \texttt{\_\_parameters\_\_} : \texttt{NoneTypeET} \to \texttt{TupleET}[\texttt{TypeVarET}, \ldots], \ldots\}$$

- `typing.Protocol` existential type:

$$\texttt{ProtocolET}_\texttt{n} = \forall T_1, \ldots, T_n.\exists P <: \texttt{GenericET}_{\texttt{n}+1}[P, T_1, \ldots, T_n].\{ \texttt{\_\_new\_\_} : \texttt{TupleET}[\texttt{ObjectET}, \ldots] \times \texttt{DictET}[\texttt{StrET}, \texttt{ObjectET}] \to \texttt{BottomET}, \texttt{\_is\_protocol} : \texttt{NoneTypeET} \to \texttt{BoolET}, \texttt{\_is\_runtime\_protocol} : \texttt{NoneTypeET} \to \texttt{Bool}, \ldots\}$$

The Foundational (Blueprint) Layer

```
class SupportsAbs[T](Protocol):
  def __abs__(self) -> T: ...
```

$$\texttt{SupportsAbsET} = \forall T.\exists SA <: \texttt{ProtocolET}_1[\texttt{T}].\{\texttt{\_\_abs\_\_} : SA \to T, \ldots\}$$

The Foundational (Blueprint) Layer

```
class SupportsAbs[T](Protocol):
  def __abs__(self) -> T: ...
```

$$\texttt{SupportsAbsET} = \forall T.\exists SA <: \texttt{ProtocolET}_1[\texttt{T}].\{\texttt{\_\_abs\_\_} : SA \to T, \ldots\}$$

```
class MyList(list):
  pretty_string = lambda self: "test"
```

$$\texttt{MyListET} = \forall T.\exists L <: \texttt{ListET}[T].\{\texttt{pretty\_string} : M \to \texttt{StrET}, \ldots\}$$

The Meta Layer

```
class MyList(list):
    pretty_string = lambda self: "test"

foo = MyList([1, 2, 3])  # MyListET[IntET]
bar = [MyList]  # a list with a single element
```

The Meta Layer

```
class MyList(list):
    pretty_string = lambda self: "test"

foo = MyList([1, 2, 3])  # MyListET[IntET]
bar = [MyList]  # a list with a single element
```

The Meta Layer

```
class MyList(list):
    pretty_string = lambda self: "test"

foo = MyList([1, 2, 3])  # MyListET[IntET]
bar = [MyList]  # a list with a single element
```

Dual role of classes:

- blueprint role: defines the structure and behavior of its instances;
- value role: a value that exists at runtime (class-as-value).

The Meta Layer

```
class MyList(list):
    pretty_string = lambda self: "test"

foo = MyList([1, 2, 3])  # MyListET[IntET]
bar = [MyList]  # a list with a single element
```

Dual role of classes:

- blueprint role: defines the structure and behavior of its instances;
- value role: a value that exists at runtime (class-as-value).

```
MyList = type('MyList', (list, ), {'pretty_string':
    lambda self: "test"})

foo = MyList([1, 2, 3])  # MyListET[IntET]
bar = [MyList]  # a list with a single element
```

The Meta Layer

```
class MyList(list):
    pretty_string = lambda self: "test"

foo = MyList([1, 2, 3])  # MyListET[IntET]
bar = [MyList]  # a list with a single element
```

Dual role of classes:

- blueprint role: defines the structure and behavior of its instances;
- value role: a value that exists at runtime (class-as-value).

```
MyList = type('MyList', (list, ), {'pretty_string':
    lambda self: "test"})

foo = MyList([1, 2, 3])  # MyListET[IntET]
bar = [MyList]  # a list with a single element
```

The Meta Layer

```
class MyList(list):
    pretty_string = lambda self: "test"

foo = MyList([1, 2, 3])  # MyListET[IntET]
bar = [MyList]  # a list with a single element
```

Dual role of classes:

- blueprint role: defines the structure and behavior of its instances;
- value role: a value that exists at runtime (class-as-value).

```
MyList = type('MyList', (list, ), {'pretty_string':
    lambda self: "test"})

foo = MyList([1, 2, 3])  # MyListET[IntET]
bar = [MyList]  # a list with a single element
```

The Meta Layer

```
class MyList(list):
    pretty_string = lambda self: "test"

foo = MyList([1, 2, 3])  # MyListET[IntET]
bar = [MyList]  # a list with a single element
```

Dual role of classes:

- blueprint role: defines the structure and behavior of its instances;
- value role: a value that exists at runtime (class-as-value).

```
MyList = type('MyList', (list, ), {'pretty_string':
    lambda self: "test"})

foo = MyList([1, 2, 3])  # MyListET[IntET]
bar = [MyList]  # a list with a single element
```

The Meta Layer

```
MyList = type('MyList', (list, ), {'pretty_string':
    lambda self: "test"})

foo = MyList([1, 2, 3])  # MyListET[IntET]
bar = [MyList]  # a list with a single element
```

$$\texttt{TypeET} = \exists M <: \texttt{ObjectET}.\{\texttt{\_\_new\_\_} : \texttt{StrET} \times \texttt{TupleET}[\texttt{TypeET}, \ldots] \times \texttt{DictET}[\texttt{StrET}, \texttt{ObjectET}] \times \texttt{DictET}[\texttt{StrET}, \texttt{ObjectET}] \to M, \ldots\} \text{ is PyTS}$$

The Meta Layer

```
MyList = type('MyList', (list,), {'pretty_string':
    lambda self: "test"})

foo = MyList([1, 2, 3])  # MyListET[IntET]
bar = [MyList]  # a list with a single element
```

$$\texttt{TypeET} = \exists M <: \texttt{ObjectET}.\{\texttt{\_\_new\_\_} : \texttt{StrET} \times \texttt{TupleET}[\texttt{TypeET}, \ldots] \times \texttt{DictET}[\texttt{StrET}, \texttt{ObjectET}] \times \texttt{DictET}[\texttt{StrET}, \texttt{ObjectET}] \to M, \ldots\} \texttt{ is PyTS}$$

- the elements of M are representations of classes-as-values;

The Meta Layer

```
MyList = type('MyList', (list,), {'pretty_string':
    lambda self: "test"})

foo = MyList([1, 2, 3])  # MyListET[IntET]
bar = [MyList]  # a list with a single element
```

$$\texttt{TypeET} = \exists M <: \texttt{ObjectET}.\{\texttt{\_\_new\_\_} : \texttt{StrET} \times \texttt{TupleET}[\texttt{TypeET}, \ldots] \times \texttt{DictET}[\texttt{StrET}, \texttt{ObjectET}] \times \texttt{DictET}[\texttt{StrET}, \texttt{ObjectET}] \to M, \ldots\} \texttt{ is PyTS}$$

- the elements of M are representations of classes-as-values;
- type is the *canonical witness* of this existential type;

The Meta Layer

```
MyList = type('MyList', (list,), {'pretty_string':
    lambda self: "test"})

foo = MyList([1, 2, 3])  # MyListET[IntET]
bar = [MyList]  # a list with a single element
```

$$\texttt{TypeET} = \exists M <: \texttt{ObjectET}.\{\texttt{\_\_new\_\_} : \texttt{StrET} \times \texttt{TupleET}[\texttt{TypeET}, \ldots] \times \texttt{DictET}[\texttt{StrET}, \texttt{ObjectET}] \times \texttt{DictET}[\texttt{StrET}, \texttt{ObjectET}] \to M, \ldots\} \texttt{ is PyTS}$$

- the elements of M are representations of classes-as-values;
- `type` is the *canonical witness* of this existential type;
- `type` is both a value of `TypeET` and the mechanism for creating other values of `TypeET`;

The Meta Layer

```
MyList = type('MyList', (list,), {'pretty_string':
    lambda self: "test"})

foo = MyList([1, 2, 3])  # MyListET[IntET]
bar = [MyList]  # a list with a single element
```

$$\texttt{TypeET} = \exists M <: \texttt{ObjectET}.\{\texttt{\_\_new\_\_} : \texttt{StrET} \times \texttt{TupleET[TypeET, ...]} \times \texttt{DictET[StrET, ObjectET]} \times \texttt{DictET[StrET, ObjectET]} \to M, ...\}\ \texttt{is PyTS}$$

- the elements of M are representations of classes-as-values;
- `type` is the *canonical witness* of this existential type;
- `type` is both a value of `TypeET` and the mechanism for creating other values of `TypeET`;
- tuple of classes-as-values, e.g. `(int, )`.

The Meta Layer

```
MyList = type('MyList', (list,), {'pretty_string':
    lambda self: "test"})

foo = MyList([1, 2, 3])  # MyListET[IntET]
bar = [MyList]  # ListET[TypeET]
```

$$\texttt{TypeET} = \exists M <: \texttt{ObjectET}.\{\texttt{\_\_new\_\_} : \texttt{StrET} \times \texttt{TupleET[TypeET, ...]} \times \texttt{DictET[StrET, ObjectET]} \times \texttt{DictET[StrET, ObjectET]} \to M, ...\}\ \texttt{is PyTS}$$

- the elements of M are representations of classes-as-values;
- `type` is the *canonical witness* of this existential type;
- `type` is both a value of `TypeET` and the mechanism for creating other values of `TypeET`;
- tuple of classes-as-values, e.g. `(int,)`.

Table of Contents

1. Introduction
   - Context
   - Type Annotations
   - Duck Typing
   - Everything is an Object
   - Classes and Metaclasses
   - ABCs and Protocols
2. Static Type System Proposal
   - Existential Types
   - Static Type System for Python
     - The Foundational (Blueprint) Layer
     - The Meta Layer
3. Conclusion and Future Work

Conclusion and Future Work

Conclusion:

Conclusion and Future Work

Conclusion:

- Presented key concepts of Python's type system, e.g. metaclasses, duck typing, Protocols and ABCs;

Conclusion and Future Work

Conclusion:

- Presented key concepts of Python's type system, e.g. metaclasses, duck typing, Protocols and ABCs;
- Established a formal typing foundation for static Python types using ADTs and existential types.

Conclusion and Future Work

Conclusion:

- Presented key concepts of Python's type system, e.g. metaclasses, duck typing, Protocols and ABCs;
- Established a formal typing foundation for static Python types using ADTs and existential types.

Future Work:

Conclusion and Future Work

Conclusion:

- Presented key concepts of Python's type system, e.g. metaclasses, duck typing, Protocols and ABCs;
- Established a formal typing foundation for static Python types using ADTs and existential types.

Future Work:

- Extend the formalism by formalizing the subtyping relation;

Conclusion and Future Work

Conclusion:

- Presented key concepts of Python's type system, e.g. metaclasses, duck typing, Protocols and ABCs;
- Established a formal typing foundation for static Python types using ADTs and existential types.

Future Work:

- Extend the formalism by formalizing the subtyping relation;
- Develop of a type inference framework;

Conclusion and Future Work

Conclusion:

- Presented key concepts of Python's type system, e.g. metaclasses, duck typing, Protocols and ABCs;
- Established a formal typing foundation for static Python types using ADTs and existential types.

Future Work:

- Extend the formalism by formalizing the subtyping relation;
- Develop of a type inference framework;
- Programmatic extraction of blueprints from stub files;

Conclusion and Future Work

Conclusion:

- Presented key concepts of Python's type system, e.g. metaclasses, duck typing, Protocols and ABCs;
- Established a formal typing foundation for static Python types using ADTs and existential types.

Future Work:

- Extend the formalism by formalizing the subtyping relation;
- Develop of a type inference framework;
- Programmatic extraction of blueprints from stub files;
- Formal soundness proof for a well-defined subset of the Python language.

Thank You!

A class holds the type,
abstract structure veiled within,
essence kept inside.