mypyc

Release 1.11.0+dev.6ebce43143c898db3de83f31b2b9e5f34e3000fa.dirty

mypyc team

Apr 26, 2024

FIRST STEPS

Mypyc compiles Python modules to C extensions. It uses standard Python type hints to generate fast code.

CHAPTER

ONE

INTRODUCTION

Mypyc compiles Python modules to C extensions. It uses standard Python type hints to generate fast code.

The compiled language is a strict, *gradually typed* Python variant. It restricts the use of some dynamic Python features to gain performance, but it's mostly compatible with standard Python.

Mypyc uses mypy to perform type checking and type inference. Most type system features in the stdlib typing module are supported.

Compiled modules can import arbitrary Python modules and third-party libraries. You can compile anything from a single performance-critical module to your entire codebase. You can run the modules you compile also as normal, interpreted Python modules.

Existing code with type annotations is often **1.5x to 5x** faster when compiled. Code tuned for mypyc can be **5x to 10x** faster.

Mypyc currently aims to speed up non-numeric code, such as server applications. Mypyc is also used to compile itself (and mypy).

1.1 Why mypyc?

Easy to get started. Compiled code has the look and feel of regular Python code. Mypyc supports familiar Python syntax and idioms.

Expressive types. Mypyc fully supports standard Python type hints. Mypyc has local type inference, generics, optional types, tuple types, union types, and more. Type hints act as machine-checked documentation, making code not only faster but also easier to understand and modify.

Python ecosystem. Mypyc runs on top of CPython, the standard Python implementation. You can use any third-party libraries, including C extensions, installed with pip. Mypyc uses only valid Python syntax, so all Python editors and IDEs work perfectly.

Fast program startup. Mypyc uses ahead-of-time compilation, so compilation does not slow down program startup. Slow program startup is a common issue with JIT compilers.

Migration path for existing code. Existing Python code often requires only minor changes to compile using mypyc.

Waiting for compilation is optional. Compiled code also runs as normal Python code. You can use interpreted Python during development, with familiar and fast workflows.

Runtime type safety. Mypyc protects you from segfaults and memory corruption. Any unexpected runtime type safety violation is a bug in mypyc. Runtime values are checked against type annotations. (Without mypyc, type annotations are ignored at runtime.)

Find errors statically. Mypyc uses mypy for static type checking that helps catch many bugs.

1.2 Use cases

Fix only performance bottlenecks. Often most time is spent in a few Python modules or functions. Add type annotations and compile these modules for easy performance gains.

Compile it all. During development you can use interpreted mode, for a quick edit-run cycle. In releases all non-test code is compiled. This is how mypy achieved a 4x performance improvement over interpreted Python.

Take advantage of existing type hints. If you already use type annotations in your code, adopting mypyc will be easier. You've already done most of the work needed to use mypyc.

Alternative to a lower-level language. Instead of writing performance-critical code in C, C++, Cython or Rust, you may get good performance while staying in the comfort of Python.

Migrate C extensions. Maintaining C extensions is not always fun for a Python developer. With mypyc you may get performance similar to the original C, with the convenience of Python.

1.3 Differences from Cython

Mypyc targets many similar use cases as Cython. Mypyc does many things differently, however:

- No need to use non-standard syntax, such as `cpdef`, or extra decorators to get good performance. Clean, normal-looking type-annotated Python code can be fast without language extensions. This makes it practical to compile entire codebases without a developer productivity hit.
- Mypyc has first-class support for features in the `typing` module, such as tuple types, union types and generics.
- Mypyc has powerful type inference, provided by mypy. Variable type annotations are not needed for optimal performance.
- Mypyc fully integrates with mypy for robust and seamless static type checking.
- Mypyc performs strict enforcement of type annotations at runtime, resulting in better runtime type safety and easier debugging.

Unlike Cython, mypyc doesn't directly support interfacing with C libraries or speeding up numeric code.

1.4 How does it work

Mypyc uses several techniques to produce fast code:

- Mypyc uses *ahead-of-time compilation* to native code. This removes CPython interpreter overhead.
- Mypyc enforces type annotations (and type comments) at runtime, raising `TypeError` if runtime values don't match annotations. Value types only need to be checked in the boundaries between dynamic and static typing.
- Compiled code uses optimized, type-specific primitives.
- Mypyc uses *early binding* to resolve called functions and name references at compile time. Mypyc avoids many dynamic namespace lookups.
- Classes are compiled to *C extension classes*. They use vtables for fast method calls and attribute access.
- Mypyc treats compiled functions, classes, and attributes declared `Final` as immutable.
- Mypyc has memory-efficient, unboxed representations for integers and booleans.

1.5 Development status

Mypyc is currently alpha software. It's only recommended for production use cases with careful testing, and if you are willing to contribute fixes or to work around issues you will encounter.

CHAPTER

TWO

GETTING STARTED

Here you will learn some basic things you need to know to get started with mypyc.

2.1 Prerequisites

You need a Python C extension development environment. The way to set this up depends on your operating system.

2.1.1 macOS

Install Xcode command line tools:

```
$ xcode-select --install
```

2.1.2 Linux

You need a C compiler and CPython headers and libraries. The specifics of how to install these varies by distribution. Here are instructions for Ubuntu 18.04, for example:

```
$ sudo apt install python3-dev
```

2.1.3 Windows

From Build Tools for Visual Studio 2022, install MSVC C++ build tools for your architecture and a Windows SDK. (latest versions recommended)

2.2 Installation

Mypyc is shipped as part of the mypy distribution. Install mypy like this (you need Python 3.8 or later):

```
$ python3 -m pip install -U 'mypy[mypyc]'
```

On some systems you need to use this instead:

```
$ python -m pip install -U 'mypy[mypyc]'
```

2.3 Example program

Let's start with a classic micro-benchmark, recursive fibonacci. Save this file as `fib.py`:

```
import time

def fib(n: int) -> int:
    if n <= 1:
        return n
    else:
        return fib(n - 2) + fib(n - 1)

t0 = time.time()
fib(32)
print(time.time() - t0)
```

Note that we gave the `fib` function a type annotation. Without it, performance won't be as impressive after compilation.

Note: Mypy documentation is a good introduction if you are new to type annotations or mypy. Mypyc uses mypy to perform type checking and type inference, so some familiarity with mypy is very useful.

2.4 Compiling and running

We can run `fib.py` as a regular, interpreted program using CPython:

```
$ python3 fib.py
0.4125328063964844
```

It took about 0.41s to run on my computer.

Run `mypyc` to compile the program to a binary C extension:

```
$ mypyc fib.py
```

This will generate a C extension for `fib` in the current working directory. For example, on a Linux system the generated file may be called `fib.cpython-37m-x86_64-linux-gnu.so`.

Since C extensions can't be run as programs, use `python3 -c` to run the compiled module as a program:

```
$ python3 -c "import fib"
0.04097270965576172
```

After compilation, the program is about 10x faster. Nice!

Note: `__name__` in `fib.py` would now be `"fib"`, not `"__main__"`.

You can also pass most mypy command line options to `mypyc`.

2.5 Deleting compiled binary

You can manually delete the C extension to get back to an interpreted version (this example works on Linux):

```
$ rm fib.*.so
```

2.6 Using setup.py

You can also use `setup.py` to compile modules using mypyc. Here is an example `setup.py` file:

```
from setuptools import setup

from mypyc.build import mypycify

setup(
    name='mylib',
    packages=['mylib'],
    ext_modules=mypycify([
        'mylib/__init__.py',
        'mylib/mod.py',
    ]),
)
```

We used `mypycify(...)` to specify which files to compile using mypyc. Your `setup.py` can include additional Python files outside `mypycify(...)` that won't be compiled.

Now you can build a wheel (.whl) file for the package:

```
python3 setup.py bdist_wheel
```

The wheel is created under `dist/`.

You can also compile the C extensions in-place, in the current directory (similar to using `mypyc` to compile modules):

```
python3 setup.py build_ext --inplace
```

You can include most mypy command line options in the list of arguments passed to `mypycify()`. For example, here we use the `--disallow-untyped-defs` flag to require that all functions have type annotations:

```
...
setup(
    name='frobnicate',
    packages=['frobnicate'],
    ext_modules=mypycify([
        '--disallow-untyped-defs',  # Pass a mypy flag
        'frobnicate.py',
    ]),
)
```

2.7 Recommended workflow

A simple way to use mypyc is to always compile your code after any code changes, but this can get tedious, especially if you have a lot of code. Instead, you can do most development in interpreted mode. This development workflow has worked smoothly for developing mypy and mypyc (often we forget that we aren't working on a vanilla Python project):

1. During development, use interpreted mode. This gives you a fast edit-run cycle.
2. Use type annotations liberally and use mypy to type check your code during development. Mypy and tests can find most errors that would break your compiled code, if you have good type annotation coverage. (Running mypy is pretty quick.)
3. After you've implemented a feature or a fix, compile your project and run tests again, now in compiled mode. Usually nothing will break here, assuming your type annotation coverage is good. This can happen locally or in a Continuous Integration (CI) job. If you have CI, compiling locally may be rarely needed.
4. Release or deploy a compiled version. Optionally, include a fallback interpreted version for platforms that mypyc doesn't support.

This mypyc workflow only involves minor tweaks to a typical Python workflow. Most of development, testing and debugging happens in interpreted mode. Incremental mypy runs, especially when using the mypy daemon, are very quick (often a few hundred milliseconds).

2.8 Next steps

You can sometimes get good results by just annotating your code and compiling it. If this isn't providing meaningful performance gains, if you have trouble getting your code to work under mypyc, or if you want to optimize your code for maximum performance, you should read the rest of the documentation in some detail.

Here are some specific recommendations, or you can just read the documentation in order:

- *Using type annotations*
- *Native classes*
- *Differences from Python*
- *Performance tips and tricks*

CHAPTER

THREE

USING TYPE ANNOTATIONS

You will get the most out of mypyc if you compile code with precise type annotations. Not all type annotations will help performance equally, however. Using types such as *primitive types*, *native classes*, *union types*, *trait types*, and *tuple types* as much as possible is a key to major performance gains over CPython.

In contrast, some other types, including `Any`, are treated as *erased types*. Operations on erased types use generic operations that work with arbitrary objects, similar to how the CPython interpreter works. If you only use erased types, the only notable benefits over CPython will be the removal of interpreter overhead (from compilation) and a bit of *early binding*, which will usually only give minor performance gains.

3.1 Primitive types

The following built-in types are treated as *primitive types* by mypyc, and many operations on these types have efficient implementations:

- `int` (*native operations*)
- `i64` (*documentation*, *native operations*)
- `i32` (*documentation*, *native operations*)
- `i16` (*documentation*, *native operations*)
- `u8` (*documentation*, *native operations*)
- `float` (*native operations*)
- `bool` (*native operations*)
- `str` (*native operations*)
- `List[T]` (*native operations*)
- `Dict[K, V]` (*native operations*)
- `Set[T]` (*native operations*)
- `Tuple[T, ...]` (variable-length tuple; *native operations*)
- `None`

The link after each type lists all supported native, optimized operations for the type. You can use all operations supported by Python, but *native operations* will have custom, optimized implementations.

3.2 Primitive containers

Primitive container objects such as `list` and `dict` don't maintain knowledge of the item types at runtime – the item type is *erased*.

This means that item types are checked when items are accessed, not when a container is passed as an argument or assigned to another variable. For example, here we have a runtime type error on the final line of `example` (the `Any` type means an arbitrary, unchecked value):

```
from typing import List, Any

def example(a: List[Any]) -> None:
    b: List[int] = a  # No error -- items are not checked
    print(b[0])  # Error here -- got str, but expected int

example(["x"])
```

3.3 Native classes

Classes that get compiled to C extensions are called native classes. Most common operations on instances of these classes are optimized, including construction, attribute access and method calls.

Native class definitions look exactly like normal Python class definitions. A class is usually native if it's in a compiled module (though there are some exceptions).

Consider this example:

```
class Point:
    def __init__(self, x: int, y: int) -> None:
        self.x = x
        self.y = y

def shift(p: Point) -> Point:
    return Point(p.x + 1, p.y + 1)
```

All operations in the above example use native operations, if the file is compiled.

Native classes have some notable different from Python classes:

- Only attributes and methods defined in the class body or methods are supported. If you try to assign to an undefined attribute outside the class definition, `AttributeError` will be raised. This enables an efficient memory layout and fast method calls for native classes.
- Native classes usually don't define the `__dict__` attribute (they don't have an attribute dictionary). This follows from only having a specific set of attributes.
- Native classes can't have an arbitrary metaclass or use most class decorators.

Native classes only support single inheritance. A limited form of multiple inheritance is supported through *trait types*. You generally must inherit from another native class (or `object`). By default, you can't inherit a Python class from a native class (but there's an *override* to allow that).

See *Native classes* for more details.

3.4 Tuple types

Fixed-length tuple types such as `Tuple[int, str]` are represented as *value types* when stored in variables, passed as arguments, or returned from functions. Value types are allocated in the low-level machine stack or in CPU registers, as opposed to *heap types*, which are allocated dynamically from the heap.

Like all value types, tuples will be *boxed*, i.e. converted to corresponding heap types, when stored in Python containers, or passed to non-native code. A boxed tuple value will be a regular Python tuple object.

3.5 Union types

Union types and optional types that contain primitive types, native class types and trait types are also efficient. If a union type has *erased* items, accessing items with non-erased types is often still quite efficient.

A value with a union types is always *boxed*, even if it contains a value that also has an unboxed representation, such as an integer or a boolean.

For example, using `Optional[int]` is quite efficient, but the value will always be boxed. A plain `int` value will usually be faster, since it has an unboxed representation.

3.6 Trait types

Trait types enable a form of multiple inheritance for native classes. A native class can inherit any number of traits. Trait types are defined as classes using the `mypy_extensions.trait` decorator:

```
from mypy_extensions import trait

@trait
class MyTrait:
    def method(self) -> None:
        ...
```

Traits can define methods, properties and attributes. They often define abstract methods. Traits can be generic.

If a class subclasses both a non-trait class and traits, the traits must be placed at the end of the base class list:

```
class Base: ...

class Derived(Base, MyTrait, FooTrait):  # OK
    ...

class Derived2(MyTrait, FooTrait, Base):
    # Error: traits should come last
    ...
```

Traits have some special properties:

- You shouldn't create instances of traits (though mypyc does not prevent it yet).
- Traits can subclass other traits or native classes, but the MRO must be linear (just like with native classes).
- Accessing methods or attributes through a trait type is somewhat less efficient than through a native class type, but this is much faster than through Python class types or other *erased types*.

You need to install `mypy-extensions` to use `@trait`:

```
pip install --upgrade mypy-extensions
```

3.7 Erased types

Mypyc supports many other kinds of types as well, beyond those described above. However, these types don't have customized operations, and they are implemented using *type erasure*. Type erasure means that all other types are equivalent to untyped values at runtime, i.e. they are the equivalent of the type `Any`. Erased types include these:

- Python classes (including ABCs)
- Non-mypyc extension types and primitive types (including built-in types that are not primitives)
- Callable types
- Type variable types
- Type Any
- Protocol types

Using erased types can still improve performance, since they can enable better types to be inferred for expressions that use these types. For example, a value with type `Callable[[], int]` will not allow native calls. However, the return type is a primitive type, and we can use fast operations on the return value:

```
from typing import Callable

def call_and_inc(f: Callable[[], int]) -> int:
    # Slow call, since f has an erased type
    n = f()
    # Fast increment; inferred type of n is int (primitive type)
    n += 1
    return n
```

If the type of the argument `f` was `Any`, the type of `n` would also be `Any`, resulting in a generic, slower increment operation being used.

3.8 Strict runtime type checking

Compiled code ensures that any variable or expression with a non-erased type only has compatible values at runtime. This is in contrast with using *optional static typing*, such as by using mypy, when type annotations are not enforced at runtime. Mypyc ensures type safety both statically and at runtime.

`Any` types and erased types in general can compromise type safety, and this is by design. Inserting strict runtime type checks for all possible values would be too expensive and against the goal of high performance.

3.9 Value and heap types

In CPython, memory for all objects is dynamically allocated on the heap. All Python types are thus *heap types*. In compiled code, some types are *value types* – no object is (necessarily) allocated on the heap. `bool`, `float`, `None`, *native integer types* and fixed-length tuples are value types.

`int` is a hybrid. For typical integer values, it is a value type. Large enough integer values, those that require more than 63 bits (or 31 bits on 32-bit platforms) to represent, use a heap-based representation (same as CPython).

Value types have a few differences from heap types:

- When an instance of a value type is used in a context that expects a heap value, for example as a list item, it will transparently switch to a heap-based representation (boxing) as needed.
- Similarly, mypyc transparently changes from a heap-based representation to a value representation (unboxing).
- Object identity of integers, floating point values and tuples is not preserved. You should use == instead of `is` if you are comparing two integers, floats or fixed-length tuples.
- When an instance of a subclass of a value type is converted to the base type, it is implicitly converted to an instance of the target type. For example, a `bool` value assigned to a variable with an `int` type will be converted to the corresponding integer.

The latter conversion is the only implicit type conversion that happens in mypyc programs.

Example:

```
def example() -> None:
    # A small integer uses the value (unboxed) representation
    x = 5
    # A large integer uses the heap (boxed) representation
    x = 2**500
    # Lists always contain boxed integers
    a = [55]
    # When reading from a list, the object is automatically unboxed
    x = a[0]
    # True is converted to 1 on assignment
    x = True
```

Since integers and floating point values have a different runtime representations and neither can represent all the values of the other type, type narrowing of floating point values through assignment is disallowed in compiled code. For consistency, mypyc rejects assigning an integer value to a float variable even in variable initialization. An explicit conversion is required.

Examples:

```
def narrowing(n: int) -> None:
    # Error: Incompatible value representations in assignment
    # (expression has type "int", variable has type "float")
    x: float = 0

    y: float = 0.0  # Ok

    if f():
        y = n  # Error
    if f():
        y = float(n)  # Ok
```

3.10 Native integer types

You can use the native integer types `i64` (64-bit signed integer), `i32` (32-bit signed integer), `i16` (16-bit signed integer), and `u8` (8-bit unsigned integer) if you know that integer values will always fit within fixed bounds. These types are faster than the arbitrary-precision `int` type, since they don't require overflow checks on operations. They may also use less memory than `int` values. The types are imported from the `mypy_extensions` module (installed via `pip install mypy_extensions`).

Example:

```
from mypy_extensions import i64

def sum_list(l: list[i64]) -> i64:
    s: i64 = 0
    for n in l:
        s += n
    return s

# Implicit conversions from int to i64
print(sum_list([1, 3, 5]))
```

Note: Since there are no overflow checks when performing native integer arithmetic, the above function could result in an overflow or other undefined behavior if the sum might not fit within 64 bits.

The behavior when running as interpreted Python program will be different if there are overflows. Declaring native integer types have no effect unless code is compiled. Native integer types are effectively equivalent to `int` when interpreted.

Native integer types have these additional properties:

- Values can be implicitly converted between `int` and a native integer type (both ways).
- Conversions between different native integer types must be explicit. A conversion to a narrower native integer type truncates the value without a runtime overflow check.
- If a binary operation (such as +) or an augmented assignment (such as +=) mixes native integer and `int` values, the `int` operand is implicitly coerced to the native integer type (native integer types are "sticky").
- You can't mix different native integer types in binary operations. Instead, convert between types explicitly.

For more information about native integer types, refer to *native integer operations*.

CHAPTER

FOUR

NATIVE CLASSES

Classes in compiled modules are *native classes* by default (some exceptions are discussed below). Native classes are compiled to C extension classes, which have some important differences from normal Python classes. Native classes are similar in many ways to built-in types, such as `int`, `str`, and `list`.

4.1 Immutable namespaces

The type object namespace of native classes is mostly immutable (but class variables can be assigned to):

```
class Cls:
    def method1(self) -> None:
        print("method1")

    def method2(self) -> None:
        print("method2")

Cls.method1 = Cls.method2  # Error
Cls.new_method = Cls.method2  # Error
```

Only attributes defined within a class definition (or in a base class) can be assigned to (similar to using `__slots__`):

```
class Cls:
    x: int

    def __init__(self, y: int) -> None:
        self.x = 0
        self.y = y

    def method(self) -> None:
        self.z = "x"

o = Cls(0)
print(o.x, o.y)  # OK
o.z = "y"  # OK
o.extra = 3  # Error: no attribute "extra"
```

4.2 Inheritance

Only single inheritance is supported (except for *traits*). Most non-native classes can't be used as base classes.

These non-native classes can be used as base classes of native classes:

- `object`
- `dict` (and `Dict[k, v]`)
- `BaseException`
- `Exception`
- `ValueError`
- `IndexError`
- `LookupError`
- `UserWarning`
- `typing.NamedTuple`
- `enum.Enum`

By default, a non-native class can't inherit a native class, and you can't inherit from a native class outside the compilation unit that defines the class. You can enable these through `mypy_extensions.mypyc_attr`:

```
from mypy_extensions import mypyc_attr

@mypyc_attr(allow_interpreted_subclasses=True)
class Cls:
    ...
```

Allowing interpreted subclasses has only minor impact on performance of instances of the native class. Accessing methods and attributes of a *non-native* subclass (or a subclass defined in another compilation unit) will be slower, since it needs to use the normal Python attribute access mechanism.

You need to install `mypy-extensions` to use `@mypyc_attr`:

```
pip install --upgrade mypy-extensions
```

4.3 Class variables

Class variables must be explicitly declared using `attr: ClassVar` or `attr: ClassVar[<type>]`. You can't assign to a class variable through an instance. Example:

```
from typing import ClassVar

class Cls:
    cv: ClassVar = 0

Cls.cv = 2  # OK
o = Cls()
print(o.cv)  # OK (2)
o.cv = 3  # Error!
```

Tip: Constant class variables can be declared using `typing.Final` or `typing.Final[<type>]`.

4.4 Generic native classes

Native classes can be generic. Type variables are *erased* at runtime, and instances don't keep track of type variable values.

Compiled code thus can't check the values of type variables when performing runtime type checks. These checks are delayed to when reading a value with a type variable type:

```
from typing import TypeVar, Generic, cast

T = TypeVar('T')

class Box(Generic[T]):
    def __init__(self, item: T) -> None:
        self.item = item

x = Box(1)  # Box[int]
y = cast(Box[str], x)  # OK (type variable value not checked)
y.item  # Runtime error: item is "int", but "str" expected
```

4.5 Metaclasses

Most metaclasses aren't supported with native classes, since their behavior is too dynamic. You can use these metaclasses, however:

- `abc.ABCMeta`
- `typing.GenericMeta` (used by `typing.Generic`)

Note: If a class definition uses an unsupported metaclass, *mypyc compiles the class into a regular Python class*.

4.6 Class decorators

Similar to metaclasses, most class decorators aren't supported with native classes, as they are usually too dynamic. These class decorators can be used with native classes, however:

- `mypy_extensions.trait` (for defining *trait types*)
- `mypy_extensions.mypyc_attr` (see *above*)
- `dataclasses.dataclass`
- `@attr.s(auto_attribs=True)`

Dataclasses and attrs classes have partial native support, and they aren't as efficient as pure native classes.

Note: If a class definition uses an unsupported class decorator, *mypyc compiles the class into a regular Python class*.

4.7 Deleting attributes

By default, attributes defined in native classes can't be deleted. You can explicitly allow certain attributes to be deleted by using `__deletable__`:

```
class Cls:
    x: int = 0
    y: int = 0
    other: int = 0

    __deletable__ = ['x', 'y']  # 'x' and 'y' can be deleted

o = Cls()
del o.x  # OK
del o.y  # OK
del o.other  # Error
```

You must initialize the `__deletable__` attribute in the class body, using a list or a tuple expression with only string literal items that refer to attributes. These are not valid:

```
a = ['x', 'y']

class Cls:
    x: int
    y: int

    __deletable__ = a  # Error: cannot use variable 'a'

__deletable__ = ('a',)  # Error: not in a class body
```

4.8 Other properties

Instances of native classes don't usually have a `__dict__` attribute.

CHAPTER

FIVE

DIFFERENCES FROM PYTHON

Mypyc aims to be sufficiently compatible with Python semantics so that migrating code to mypyc often doesn't require major code changes. There are various differences to enable performance gains that you need to be aware of, however.

This section documents notable differences from Python. We discuss many of them also elsewhere, but it's convenient to have them here in one place.

5.1 Running compiled modules

You can't use `python3 <module>.py` or `python3 -m <module>` to run compiled modules. Use `python3 -c "import <module>"` instead, or write a wrapper script that imports your module.

As a side effect, you can't rely on checking the `__name__` attribute in compiled code, like this:

```
if __name__ == "__main__":  # Can't be used in compiled code
    main()
```

5.2 Type errors prevent compilation

You can't compile code that generates mypy type check errors. You can sometimes ignore these with a `# type: ignore` comment, but this can result in bad code being generated, and it's considered dangerous.

Note: In the future, mypyc may reject `# type: ignore` comments that may be unsafe.

5.3 Runtime type checking

Non-erased types in annotations will be type checked at runtime. For example, consider this function:

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

If you try to call this function with a `float` or `str` argument, you'll get a type error on the call site, even if the call site is not being type checked:

```
twice(5)  # OK
twice(2.2)  # TypeError
twice("blah")  # TypeError
```

Also, values with *inferred* types will be type checked. For example, consider a call to the stdlib function `socket.gethostname()` in compiled code. This function is not compiled (no stdlib modules are compiled with mypyc), but mypyc uses a *library stub file* to infer the return type as `str`. Compiled code calling `gethostname()` will fail with `TypeError` if `gethostname()` would return an incompatible value, such as `None`:

```
import socket

# Fail if returned value is not a str
name = socket.gethostname()
```

Note that `gethostname()` is defined like this in the stub file for `socket` (in typeshed):

```
def gethostname() -> str: ...
```

Thus mypyc verifies that library stub files and annotations in non-compiled code match runtime values. This adds an extra layer of type safety.

Casts such as `cast(str, x)` will also result in strict type checks. Consider this example:

```
from typing import cast
...
x = cast(str, y)
```

The last line is essentially equivalent to this Python code when compiled:

```
if not isinstance(y, str):
    raise TypeError(...)
x = y
```

In interpreted mode `cast` does not perform a runtime type check.

5.4 Native classes

Native classes behave differently from Python classes. See *Native classes* for the details.

5.5 Primitive types

Some primitive types behave differently in compiled code to improve performance.

`int` objects use an unboxed (non-heap-allocated) representation for small integer values. A side effect of this is that the exact runtime type of `int` values is lost. For example, consider this simple function:

```
def first_int(x: List[int]) -> int:
    return x[0]

print(first_int([True]))  # Output is 1, instead of True!
```

`bool` is a subclass of `int`, so the above code is valid. However, when the list value is converted to `int`, `True` is converted to the corresponding `int` value, which is 1.

Note that integers still have an arbitrary precision in compiled code, similar to normal Python integers.

Fixed-length tuples are unboxed, similar to integers. The exact type and identity of fixed-length tuples is not preserved, and you can't reliably use `is` checks to compare tuples that are used in compiled code.

5.6 Early binding

References to functions, types, most attributes, and methods in the same *compilation unit* use *early binding*: the target of the reference is decided at compile time, whenever possible. This contrasts with normal Python behavior of *late binding*, where the target is found by a namespace lookup at runtime. Omitting these namespace lookups improves performance, but some Python idioms don't work without changes.

Note that non-final module-level variables still use late binding. You may want to avoid these in very performance-critical code.

Examples of early and late binding:

```
from typing import Final

import lib  # "lib" is not compiled

x = 0
y: Final = 1

def func() -> None:
    pass

class Cls:
    def __init__(self, attr: int) -> None:
        self.attr = attr

    def method(self) -> None:
        pass

def example() -> None:
    # Early binding:
    var = y
    func()
    o = Cls()
    o.x
    o.method()

    # Late binding:
    var = x  # Module-level variable
    lib.func()  # Accessing library that is not compiled
```

5.7 Pickling and copying objects

Mypyc tries to enforce that instances native classes are properly initialized by calling `__init__` implicitly when constructing objects, even if objects are constructed through `pickle`, `copy.copy` or `copy.deepcopy`, for example.

If a native class doesn't support calling `__init__` without arguments, you can't pickle or copy instances of the class. Use the `mypy_extensions.mypyc_attr` class decorator to override this behavior and enable pickling through the `serializable` flag:

```
from mypy_extensions import mypyc_attr
import pickle

@mypyc_attr(serializable=True)
class Cls:
    def __init__(self, n: int) -> None:
        self.n = n

data = pickle.dumps(Cls(5))
obj = pickle.loads(data)  # OK
```

Additional notes:

- All subclasses inherit the `serializable` flag.
- If a class has the `allow_interpreted_subclasses` attribute, it implicitly supports serialization.
- Enabling serialization may slow down attribute access, since compiled code has to be always prepared to raise `AttributeError` in case an attribute is not defined at runtime.
- If you try to pickle an object without setting the `serializable` flag, you'll get a `TypeError` about missing arguments to `__init__`.

5.8 Monkey patching

Since mypyc function and class definitions are immutable, you can't perform arbitrary monkey patching, such as replacing functions or methods with mocks in tests.

Note: Each compiled module has a Python namespace that is initialized to point to compiled functions and type objects. This namespace is a regular `dict` object, and it *can* be modified. However, compiled code generally doesn't use this namespace, so any changes will only be visible to non-compiled code.

5.9 Stack overflows

Compiled code currently doesn't check for stack overflows. Your program may crash in an unrecoverable fashion if you have too many nested function calls, typically due to out-of-control recursion.

Note: This limitation will be fixed in the future.

5.10 Final values

Compiled code replaces a reference to an attribute declared `Final` with the value of the attribute computed at compile time. This is an example of *early binding*. Example:

```
MAX: Final = 100

def limit_to_max(x: int) -> int:
    if x > MAX:
        return MAX
    return x
```

The two references to `MAX` don't involve any module namespace lookups, and are equivalent to this code:

```
def limit_to_max(x: int) -> int:
    if x > 100:
        return 100
    return x
```

When run as interpreted, the first example will execute slower due to the extra namespace lookups. In interpreted code final attributes can also be modified.

5.11 Unsupported features

Some Python features are not supported by mypyc (yet). They can't be used in compiled code, or there are some limitations. You can partially work around some of these limitations by running your code in interpreted mode.

5.11.1 Nested classes

Nested classes are not supported.

5.11.2 Conditional functions or classes

Function and class definitions guarded by an if-statement are not supported.

5.11.3 Dunder methods

Native classes **cannot** use these dunders. If defined, they will not work as expected.

- `__del__`
- `__index__`
- `__getattr__`, `__getattribute__`
- `__setattr__`
- `__delattr__`

5.11.4 Generator expressions

Generator expressions are not supported. To make it easier to compile existing code, they are implicitly replaced with list comprehensions. *This does not always produce the same behavior.*

To work around this limitation, you can usually use a generator function instead. You can sometimes replace the generator expression with an explicit list comprehension.

5.11.5 Descriptors

Native classes can't contain arbitrary descriptors. Properties, static methods and class methods are supported.

5.11.6 Introspection

Various methods of introspection may break by using mypyc. Here's an non-exhaustive list of what won't work:

- Instance `__annotations__` is usually not kept
- Frames of compiled functions can't be inspected using `inspect`
- Compiled methods aren't considered methods by `inspect.ismethod`
- `inspect.signature` chokes on compiled functions

5.11.7 Profiling hooks and tracing

Compiled functions don't trigger profiling and tracing hooks, such as when using the `profile`, `cProfile`, or `trace` modules.

5.11.8 Debuggers

You can't set breakpoints in compiled functions or step through compiled functions using `pdb`. Often you can debug your code in interpreted mode instead.

CHAPTER

SIX

COMPILATION UNITS

When you run mypyc to compile a set of modules, these modules form a *compilation unit*. Mypyc will use early binding for references within the compilation unit.

If you run mypyc multiple times to compile multiple sets of modules, each invocation will result in a new compilation unit. References between separate compilation units will fall back to late binding, i.e. looking up names using Python namespace dictionaries. Also, all calls will use the slower Python calling convention, where arguments and the return value will be boxed (and potentially unboxed again in the called function).

For maximal performance, minimize interactions across compilation units. The simplest way to achieve this is to compile your entire program as a single compilation unit.

CHAPTER

SEVEN

MISCELLANEOUS NATIVE OPERATIONS

This is a list of various non-type-specific operations that have custom native implementations. If an operation has no native implementation, mypyc will use fallback generic implementations that are often not as fast.

Note: Operations specific to various primitive types are described in the following sections.

7.1 Operators

- `x is y` (this is very fast for all types)

7.2 Functions

- `isinstance(obj, type: type)`
- `isinstance(obj, type: tuple)`
- `cast(<type>, obj)`
- `type(obj)`
- `len(obj)`
- `abs(obj)`
- `id(obj)`
- `iter(obj)`
- `next(iter: Iterator)`
- `hash(obj)`
- `getattr(obj, attr)`
- `getattr(obj, attr, default)`
- `setattr(obj, attr, value)`
- `hasattr(obj, attr)`
- `delattr(obj, name)`
- `slice(start, stop, step)`
- `globals()`

7.3 Method decorators

- `@property`
- `@staticmethod`
- `@classmethod`
- `@abc.abstractmethod`

7.4 Statements

These variants of statements have custom implementations:

- `for ... in seq:` (for loop over a sequence)
- `for ... in enumerate(...):`
- `for ... in zip(...):`

CHAPTER

EIGHT

NATIVE INTEGER OPERATIONS

Mypyc supports these integer types:

- `int` (arbitrary-precision integer)
- `i64` (64-bit signed integer)
- `i32` (32-bit signed integer)
- `i16` (16-bit signed integer)
- `u8` (8-bit unsigned integer)

`i64`, `i32`, `i16` and `u8` are *native integer types* and are available in the `mypy_extensions` module. `int` corresponds to the Python `int` type, but uses a more efficient runtime representation (tagged pointer). Native integer types are value types.

All integer types have optimized primitive operations, but the native integer types are more efficient than `int`, since they don't require range or bounds checks.

Operations on integers that are listed here have fast, optimized implementations. Other integer operations use generic implementations that are generally slower. Some operations involving integers and other types, such as list indexing, are documented elsewhere.

8.1 Construction

`int` type:

- Integer literal
- `int(x: float)`
- `int(x: i64)`
- `int(x: i32)`
- `int(x: i16)`
- `int(x: u8)`
- `int(x: str)`
- `int(x: str, base: int)`
- `int(x: int)` (no-op)

`i64` type:

- `i64(x: int)`

- `i64(x: float)`
- `i64(x: i64)` (no-op)
- `i64(x: i32)`
- `i64(x: i16)`
- `i64(x: u8)`
- `i64(x: str)`
- `i64(x: str, base: int)`

`i32` type:

- `i32(x: int)`
- `i32(x: float)`
- `i32(x: i64)` (truncate)
- `i32(x: i32)` (no-op)
- `i32(x: i16)`
- `i32(x: u8)`
- `i32(x: str)`
- `i32(x: str, base: int)`

`i16` type:

- `i16(x: int)`
- `i16(x: float)`
- `i16(x: i64)` (truncate)
- `i16(x: i32)` (truncate)
- `i16(x: i16)` (no-op)
- `i16(x: u8)`
- `i16(x: str)`
- `i16(x: str, base: int)`

Conversions from `int` to a native integer type raise `OverflowError` if the value is too large or small. Conversions from a wider native integer type to a narrower one truncate the value and never fail. More generally, operations between native integer types don't check for overflow.

8.2 Implicit conversions

`int` values can be implicitly converted to a native integer type, for convenience. This means that these are equivalent:

```
from mypy_extensions import i64

def implicit() -> None:
    # Implicit conversion of 0 (int) to i64
    x: i64 = 0
```

(continues on next page)

(continued from previous page)

```
def explicit() -> None:
    # Explicit conversion of 0 (int) to i64
    x = i64(0)
```

Similarly, a native integer value can be implicitly converted to an arbitrary-precision integer. These two functions are equivalent:

```
def implicit(x: i64) -> int:
    # Implicit conversion from i64 to int
    return x

def explicit(x: i64) -> int:
    # Explicit conversion from i64 to int
    return int(x)
```

8.3 Operators

- Arithmetic (+, -, *, //, /, %)
- Bitwise operations (&, |, ^, <<, >>, ~)
- Comparisons (==, !=, <, etc.)
- Augmented assignment (`x += y`, etc.)

If one of the above native integer operations overflows or underflows with signed operands, the behavior is undefined. Signed native integer types should only be used if all possible values are small enough for the type. For this reason, the arbitrary-precision `int` type is recommended for signed values unless the performance of integer operations is critical.

Operations on unsigned integers (`u8`) wrap around on overflow.

It's a compile-time error to mix different native integer types in a binary operation such as addition. An explicit conversion is required:

```
from mypy_extensions import i64, i32

def add(x: i64, y: i32) -> None:
    a = x + y  # Error (i64 + i32)
    b = x + i64(y)  # OK
```

You can freely mix a native integer value and an arbitrary-precision `int` value in an operation. The native integer type is "sticky" and the `int` operand is coerced to the native integer type:

```
def example(x: i64, y: int) -> None:
    a = x * y
    # Type of "a" is "i64"
    ...
    b = 1 - x
    # Similarly, type of "b" is "i64"
```

8.4 Statements

For loop over a range is compiled efficiently, if the `range(...)` object is constructed in the for statement (after `in`):

- `for x in range(end)`
- `for x in range(start, end)`
- `for x in range(start, end, step)`

If one of the arguments to `range` in a for loop is a native integer type, the type of the loop variable is inferred to have this native integer type, instead of `int`:

```
for x in range(i64(n)):
    # Type of "x" is "i64"
    ...
```

CHAPTER

NINE

NATIVE BOOLEAN OPERATIONS

Operations on `bool` values that are listed here have fast, optimized implementations.

9.1 Construction

- `True`
- `False`
- `bool(obj)`

9.2 Operators

- `b1 and b2`
- `b1 or b2`
- `not b`

9.3 Functions

- `any(expr for ... in ...)`
- `all(expr for ... in ...)`

CHAPTER

TEN

NATIVE FLOAT OPERATIONS

These `float` operations have fast, optimized implementations. Other floating point operations use generic implementations that are often slower.

10.1 Construction

- Float literal
- `float(x: int)`
- `float(x: i64)`
- `float(x: i32)`
- `float(x: i16)`
- `float(x: u8)`
- `float(x: str)`
- `float(x: float)` (no-op)

10.2 Operators

- Arithmetic (+, -, *, /, //, %)
- Comparisons (==, !=, <, etc.)
- Augmented assignment (`x += y`, etc.)

10.3 Functions

- `int(f)`
- `i64(f)` (convert to 64-bit signed integer)
- `i32(f)` (convert to 32-bit signed integer)
- `i16(f)` (convert to 16-bit signed integer)
- `u8(f)` (convert to 8-bit unsigned integer)
- `abs(f)`

- `math.sin(f)`
- `math.cos(f)`
- `math.tan(f)`
- `math.sqrt(f)`
- `math.exp(f)`
- `math.log(f)`
- `math.floor(f)`
- `math.ceil(f)`
- `math.fabs(f)`
- `math.pow(x, y)`
- `math.copysign(x, y)`
- `math.isinf(f)`
- `math.isnan(f)`

CHAPTER

ELEVEN

NATIVE STRING OPERATIONS

These `str` operations have fast, optimized implementations. Other string operations use generic implementations that are often slower.

11.1 Construction

- String literal
- `str(x: int)`
- `str(x: object)`

11.2 Operators

- Concatenation (`s1 + s2`)
- Indexing (`s[n]`)
- Slicing (`s[n:m]`, `s[n:]`, `s[:m]`)
- Comparisons (==, !=)
- Augmented assignment (`s1 += s2`)

11.3 Methods

- `s1.endswith(s2: str)`
- `s.join(x: Iterable)`
- `s.replace(old: str, new: str)`
- `s.replace(old: str, new: str, count: int)`
- `s.split()`
- `s.split(sep: str)`
- `s.split(sep: str, maxsplit: int)`
- `s1.startswith(s2: str)`

CHAPTER

TWELVE

NATIVE LIST OPERATIONS

These `list` operations have fast, optimized implementations. Other list operations use generic implementations that are often slower.

12.1 Construction

Construct list with specific items:

- `[item0, ..., itemN]`

Construct empty list:

- `[]`
- `list()`

Construct list from iterable:

- `list(x: Iterable)`

List comprehensions:

- `[... for ... in ...]`
- `[... for ... in ... if ...]`

12.2 Operators

- `lst[n]` (get item by integer index)
- `lst[n:m]`, `lst[n:]`, `lst[:m]`, `lst[:]` (slicing)
- `lst * n`, `n * lst`
- `obj in lst`

12.3 Statements

Set item by integer index:

- `lst[n] = x`

For loop over a list:

- `for item in lst:`

12.4 Methods

- `lst.append(obj)`
- `lst.extend(x: Iterable)`
- `lst.insert(index, obj)`
- `lst.pop(index=-1)`
- `lst.remove(obj)`
- `lst.count(obj)`
- `lst.index(obj)`
- `lst.reverse()`
- `lst.sort()`

12.5 Functions

- `len(lst: list)`

CHAPTER

THIRTEEN

NATIVE DICT OPERATIONS

These `dict` operations have fast, optimized implementations. Other dictionary operations use generic implementations that are often slower.

13.1 Construction

Construct dict from keys and values:

- `{key: value, ...}`

Construct empty dict:

- `{}`
- `dict()`

Construct dict from another object:

- `dict(d: dict)`
- `dict(x: Iterable)`

Dict comprehensions:

- `{...: ... for ... in ...}`
- `{...: ... for ... in ... if ...}`

13.2 Operators

- `d[key]`
- `value in d`

13.3 Statements

- `d[key] = value`
- `for key in d:`

13.4 Methods

- `d.get(key)`
- `d.get(key, default)`
- `d.keys()`
- `d.values()`
- `d.items()`
- `d.copy()`
- `d.clear()`
- `d1.update(d2: dict)`
- `d.update(x: Iterable)`

13.5 Functions

- `len(d: dict)`

CHAPTER

FOURTEEN

NATIVE SET OPERATIONS

These `set` operations have fast, optimized implementations. Other set operations use generic implementations that are often slower.

14.1 Construction

Construct set with specific items:

- `{item0, ..., itemN}`

Construct empty set:

- `set()`

Construct set from iterable:

- `set(x: Iterable)`

Set comprehensions:

- `{... for ... in ...}`
- `{... for ... in ... if ...}`

14.2 Operators

- `item in s`

14.3 Methods

- `s.add(item)`
- `s.remove(item)`
- `s.discard(item)`
- `s.update(x: Iterable)`
- `s.clear()`
- `s.pop()`

14.4 Functions

- `len(s: set)`

CHAPTER

FIFTEEN

NATIVE TUPLE OPERATIONS

These `tuple` operations have fast, optimized implementations. Other tuple operations use generic implementations that are often slower.

Unless mentioned otherwise, these operations apply to both fixed-length tuples and variable-length tuples.

15.1 Construction

- `item0, ..., itemN` (construct a tuple)
- `tuple(lst: list)` (construct a variable-length tuple)
- `tuple(lst: Iterable)` (construct a variable-length tuple)

15.2 Operators

- `tup[n]` (integer index)
- `tup[n:m]`, `tup[n:]`, `tup[:m]` (slicing)

15.3 Statements

- `item0, ..., itemN = tup` (for fixed-length tuples)

15.4 Functions

- `len(tup: tuple)`

CHAPTER

SIXTEEN

PERFORMANCE TIPS AND TRICKS

Performance optimization is part art, part science. Just using mypyc in a simple manner will likely make your code faster, but squeezing the most performance out of your code requires the use of some techniques we'll summarize below.

16.1 Profiling

If you are speeding up existing code, understanding where time is spent is important. Mypyc speeds up code that you compile. If most of the time is spent elsewhere, you may come back disappointed. For example, if you spend 40% of time outside compiled code, even if compiled code would go 100x faster, overall performance will only be 2.5x faster.

A simple (but often effective) approach is to record the time in various points of program execution using `time.time()`, and to print out elapsed time (or to write it to a log file).

The stdlib modules `profile` and `cProfile` can provide much more detailed data. (But these only work well with non-compiled code.)

16.2 Avoiding slow libraries

If profiling indicates that a lot of time is spent in the stdlib or third-party libraries, you still have several options.

First, if most time is spent in a few library features, you can perhaps easily reimplement them in type-annotated Python, or extract the relevant code and annotate it. Now it may be easy to compile this code to speed it up.

Second, you may be able to avoid the library altogether, or use an alternative, more efficient library to achieve the same purpose.

16.3 Type annotations

As discussed earlier, type annotations are key to major performance gains. You should at least consider adding annotations to any performance-critical functions and classes. It may also be helpful to annotate code called by this code, even if it's not compiled, since this may help mypy infer better types in the compile code. If you use libraries, ensure they have stub files with decent type annotation coverage. Writing a stub file is often easy, and you only need to annotate features you use a lot.

If annotating external code or writing stubs feel too burdensome, a simple workaround is to annotate variables explicitly. For example, here we call `acme.get_items()`, but it has no type annotation. We can use an explicit type annotation for the variable to which we assign the result:

```
from typing import List, Tuple
import acme

def work() -> None:
    # Annotate "items" to help mypyc
    items: List[Tuple[int, str]] = acme.get_items()
    for item in items:
        ...  # Do some work here
```

Without the annotation on `items`, the type would be `Any` (since `acme` has no type annotation), resulting in slower, generic operations being used later in the function.

16.4 Avoiding slow Python features

Mypyc can optimize some features more effectively than others. Here the difference is sometimes big – some things only get marginally faster at best, while others can get 10x faster, or more. Avoiding these slow features in performance-critical parts of your code can help a lot.

These are some of the most important things to avoid:

- Using class decorators or metaclasses in compiled code (that aren't properly supported by mypyc)
- Heavy reliance on interpreted Python libraries (C extensions are usually fine)

These things also tend to be relatively slow:

- Using Python classes and instances of Python classes (native classes are much faster)
- Calling decorated functions (`@property`, `@staticmethod`, and `@classmethod` are special cased and thus fast)
- Calling nested functions
- Calling functions or methods defined in other compilation units
- Using `*args` or `**kwargs`
- Using generator functions
- Using callable values (i.e. not leveraging early binding to call functions or methods)

Nested functions can often be replaced with module-level functions or methods of native classes.

Callable values and nested functions can sometimes be replaced with an instance of a native class with a single method only, such as `call(...)`. You can derive the class from an ABC, if there are multiple possible functions.

Note: Some slow features will likely get efficient implementations in the future. You should check this section every once in a while to see if some additional operations are fast.

16.5 Using fast native features

Some native operations are particularly quick relative to the corresponding interpreted operations. Using them as much as possible may allow you to see 10x or more in performance gains.

Some things are not much (or any) faster in compiled code, such as set math operations. In contrast, calling a method of a native class is much faster in compiled code.

If you are used to optimizing for CPython, you might have replaced some class instances with dictionaries, as they can be faster. However, in compiled code, this "optimization" would likely slow down your code.

Similarly, caching a frequently called method in a local variable can help in CPython, but it can slow things down in compiled code, since the code won't use *early binding*:

```
def squares(n: int) -> List[int]:
    a = []
    append = a.append  # Not a good idea in compiled code!
    for i in range(n):
        append(i * i)
    return a
```

Here are examples of features that are fast, in no particular order (this list is *not* exhaustive):

- Calling compiled functions directly defined in the same compilation unit (with positional and/or keyword arguments)
- Calling methods of native classes defined in the same compilation unit (with positional and/or keyword arguments)
- Many integer operations
- Many `float` operations
- Booleans
- *Native list operations*, such as indexing, `append`, and list comprehensions
- While loops
- For loops over ranges and lists, and with `enumerate` or `zip`
- Reading dictionary items
- `isinstance()` checks against native classes and instances of primitive types (and unions of them)
- Accessing local variables
- Accessing attributes of native classes
- Accessing final module-level attributes
- Comparing strings for equality

These features are also fast, but somewhat less so (relative to other related operations):

- Constructing instances of native classes
- Constructing dictionaries
- Setting dictionary items
- Native *dict* and *set* operations
- Accessing module-level variables

Generally anything documented as a native operation is fast, even if it's not explicitly mentioned here

16.6 Adjusting garbage collection

Compilation does not speed up cyclic garbage collection. If everything else gets much faster, it's possible that garbage collection will take a big fraction of time. You can use `gc.set_threshold()` to adjust the garbage collector to run less often:

```
import gc

# Spend less time in gc; do this before significant computation
gc.set_threshold(150000)

...  # Actual work happens here
```

16.7 Fast interpreter shutdown

If you allocate many objects, it's possible that your program spends a lot of time cleaning up when the Python runtime shuts down. Mypyc won't speed up the shutdown of a Python process much.

You can call `os._exit(code)` to immediately terminate the Python process, skipping normal cleanup. This can give a nice boost to a batch process or a command-line tool.

Note: This can be dangerous and can lose data. You need to ensure that all streams are flushed and everything is otherwise cleaned up properly.

16.8 Work smarter

Usually there are many things you can do to improve performance, even if most tweaks will yield only minor gains. The key to being effective is to focus on things that give a large gain with a small effort.

For example, low-level optimizations, such as avoiding a nested function, can be pointless, if you could instead avoid a metaclass – to allow a key class to be compiled as a native class. The latter optimization could speed up numerous method calls and attribute accesses, just like that.

CHAPTER

SEVENTEEN

INDICES AND TABLES

- genindex
- modindex
- search