MonkeyType Documentation

Release 20.4.2

Matt Page & Carl Meyer

Apr 04, 2020

Contents

MonkeyType collects runtime types of function arguments and return values, and can automatically generate stub files or even add draft type annotations directly to your Python code based on the types collected at runtime.

CHAPTER 1

Example

Say `some/module.py` originally contains:

```
def add(a, b):
    return a + b
```

And `myscript.py` contains:

```
from some.module import add

add(1, 2)
```

Now we want to infer the type annotation of `add` in `some/module.py` by running `myscript.py` with `MonkeyType`. One way is to run:

```
$ monkeytype run myscript.py
```

By default, this will dump call traces into a SQLite database in the file `monkeytype.sqlite3` in the current working directory. You can then use the `monkeytype` command to generate a stub file for a module, or apply the type annotations directly to your code.

Running `monkeytype stub some.module` will output a stub:

```
def add(a: int, b: int) -> int: ...
```

Running `monkeytype apply some.module` will modify `some/module.py` to:

```
def add(a: int, b: int) -> int:
    return a + b
```

This example demonstrates both the value and the limitations of MonkeyType. With MonkeyType, it's very easy to add annotations that reflect the concrete types you use at runtime, but those annotations may not always match the full intended capability of the functions. For instance, `add` is capable of handling many more types than just integers. Similarly, MonkeyType may generate a concrete `List` annotation where an abstract `Sequence` or `Iterable` would be more appropriate. MonkeyType's annotations are an informative first draft, to be checked and corrected by a developer.

CHAPTER 2

Motivation

Readability and static analysis are the primary motivations for adding type annotations to code. It's already common in many Python style guides to document the argument and return types for a function in its docstring; annotations are a standardized way to provide this documentation, which also permits static analysis by a typechecker such as mypy.

For more on the motivation and design of Python type annotations, see **PEP 483** and **PEP 484**.

CHAPTER 3

Requirements

MonkeyType requires Python 3.6+ and the libcst library (for applying type stubs to code files). It generates only Python 3 type annotations (no type comments).

CHAPTER 4

Installing

Install MonkeyType with pip:

```
pip install MonkeyType
```

CHAPTER 5

How MonkeyType works

MonkeyType uses the sys.setprofile hook provided by Python to interpose on function calls, function returns, and generator yields, and record the types of arguments / return values / yield values.

It generates stub files based on that data, and can use libcst to apply those stub files directly to your code.

CHAPTER 6

The Details

6.1 Configuration

Most of the useful ways to configure MonkeyType require writing Python code to implement your preferred behavior, so MonkeyType's configuration is done in Python code. To customize MonkeyType, you:

1. subclass *monkeytype.config.Config* or *monkeytype.config.DefaultConfig*,
2. override one or more methods in your subclass,
3. instantiate your subclass, and
4. point MonkeyType to your custom *Config* instance.

Let's look at those steps in more detail.

6.1.1 Subclassing `Config` or `DefaultConfig`

class monkeytype.config.**Config**

`Config` is the "empty" config; it's not usable out of the box, and requires your subclass to fill in some blanks in order to get useful behavior. It has the following methods:

trace_store() → CallTraceStore
Return the *CallTraceStore* subclass you want to use to store your call traces.

This is the one method you must override if you subclass the empty `Config`.

trace_logger() → CallTraceLogger
Return the *CallTraceLogger* subclass you want to use to log your call traces.

If you don't override, this returns an instance of *CallTraceStoreLogger* initialized with your *trace_store()*.

code_filter() → CodeFilter
Return the *code filter* that categorizes traced functions into ones you are interested in (so their traces should be stored) and ones you aren't (their traces will be ignored).

If you don't override, returns `None`, meaning all traces will be stored. This will probably include a lot of standard-library and third-party functions!

sample_rate() → int
Return the integer sampling rate for your logged call traces. If you return an integer N from this method, 1/N function calls will be traced and logged.

If you don't override, returns `None`, which disables sampling; all function calls will be traced and logged.

type_rewriter() → TypeRewriter
Return the *TypeRewriter* which will be applied to all your types when stubs are generated.

If you don't override, returns *NoOpRewriter*, which doesn't rewrite any types.

query_limit() → int
The maximum number of call traces to query from the trace store when generating stubs. If you have recorded a lot of traces, increasing this limit may improve stub accuracy, at the cost of slower stub generation.

On the other hand, if some of your recorded traces are out of date because the code has changed, and you haven't purged your trace store, increasing this limit could make stubs worse by including more outdated traces.

Defaults to 2000.

cli_context(*command: str*) → Iterator[None]
A context manager which wraps the execution of the CLI command.

MonkeyType has to import your code in order to generate stubs for it. In some cases, like if you're using Django, setup is required before your code can be imported. Use this method to define the necessary setup or teardown for your codebase.

This method must return a context manager instance. In most cases, the simplest way to do this will be with the contextlib.contextmanager decorator. For example, if you run MonkeyType against a Django codebase, you can setup Django before the command runs:

```
@contextmanager
def cli_context(self, command: str) -> Iterator[None]:
    import django
    django.setup()
    yield
```

`command` is the name of the command passed to the monkeytype cli: `'run'`, `'apply'`, etc.

The default implementation of this method returns a no-op context manager.

max_typed_dict_size() → int
The maximum size of string-keyed dictionary for which per-key value types will be stored, and (if the traced keys and value types are consistent), a *TypedDict* will be emitted instead of *Dict*. Return *0* to disable per-key type tracking and TypedDict generation.

Defaults to 0.

class monkeytype.config.**DefaultConfig**
`DefaultConfig` is the config MonkeyType uses if you don't provide your own; it's usable as-is, and you can inherit it if you just want to make some tweaks to the default setup. `DefaultConfig` overrides the following methods from *Config*:

trace_store() → SQLiteStore
Returns an instance of *SQLiteStore*, which stores call traces in a local SQLite database, by default

in the file `monkeytype.sqlite3` in the current directory. You can override the path to the SQLite database by setting the `MT_DB_PATH` environment variable.

code_filter() → CodeFilter
Returns the default code filter predicate function. If an environment variable `MONKEYTYPE_TRACE_MODULES` is defined with one or more comma separated package and/or module names, the default code filter traces only functions within the listed modules. Otherwise the default filter excludes code in the Python standard library and installed site-packages, and traces all other functions.

type_rewriter() → ChainedRewriter
Returns an instance of *ChainedRewriter* initialized with the *RemoveEmptyContainers*, *RewriteConfigDict*, and *RewriteLargeUnion* type rewriters.

6.1.2 Using your custom config subclass

Once you've written a *Config* or *DefaultConfig* subclass, you need to instantiate it and point MonkeyType to that instance. The easiest way to do this is to create a file named `monkeytype_config.py` and create a *Config* instance in it named `CONFIG`; MonkeyType will find and use this config automatically.

For example, let's say you mostly like the default config, but you want to add a sampling rate, so you put this code in the file `monkeytype_config.py`:

```
from monkeytype.config import DefaultConfig

class MyConfig(DefaultConfig):
    def sample_rate(self):
        return 1000

CONFIG = MyConfig()
```

MonkeyType will automatically find and use this config (as long as `monkeytype_config.py` is on the Python path).

Specifying a config

You can also explicitly specify the config instance to use. For instance, when tracing calls using the *monkeytype.trace()* context manager, you can just pass your config object to it:

```
from monkeytype import trace
from some.module import my_config

with trace(my_config):
    # ... run some code you want to trace here ...
```

When running *the command line utility*, use the `--config` or `-c` option to point MonkeyType to your config, e.g.:

```
$ monkeytype -c some.module:my_config stub some.module
```

6.2 Tracing function calls

The core data type in MonkeyType is the *CallTrace*. A *CallTrace* instance represents a single traced call of a single function or method, including the concrete type of each argument and the return or yield type.

A *CallTrace* is recorded by *monkeytype run* or the *trace()* context manager (or direct use of a *CallTracer*), logged via a *CallTraceLogger*, probably stored in a *CallTraceStore*, and later queried from that store by *monkeytype stub* or *monkeytype apply* and combined with all other traces of the same function in order to generate a stub or type annotation for that function.

6.2.1 monkeytype run

The simplest way to trace some function calls with MonkeyType is to run a Python script under MonkeyType tracing using `monkeytype run` or `monkeytype run -m` at the command line:

```
$ monkeytype run myscript.py
$ monkeytype run -m mymodule
```

`monkeytype run` accepts the same *monkeytype -c* option as `monkeytype stub` and `monkeytype apply`, to point MonkeyType to the config it should use.

Because of the way Python treats scripts and imported modules differently, MonkeyType will not record traces for the entry point itself (that is, the script passed to `monkeytype run` or the module passed to `run -m`); traces are recorded only for imported modules. If you want to annotate the entry point script/module, write another short script that imports and calls its function(s), and run that script with `monkeytype run`.

6.2.2 trace context manager

You can also trace calls by wrapping a section of code inside the *trace()* context manager:

```
import monkeytype

with monkeytype.trace():
    # argument and yield/return types for all function calls will be traced
    # and stored to `monkeytype.sqlite3`
```

You can pass a *Config* object to *trace()* to customize its behavior:

```
import monkeytype
from my_mt_config import my_config

with monkeytype.trace(my_config):
    # arg and yield/return types for function calls here will be traced and
    # logged as specified by your config.
```

monkeytype.**trace**([*config: Config*]) → ContextManager

Trace all enclosed function calls and log them per the given `config`. If no config is given, use the *DefaultConfig*.

6.2.3 CallTracer

class monkeytype.tracing.**CallTracer**(*logger: CallTraceLogger*, *code_filter: CodeFilter*, *sample_rate: int*)

For more complex tracing cases where you can't easily wrap the code to trace in a context manager, you can also use a *CallTracer* directly. *CallTracer* doesn't accept a *Config* object; instead you pass it a logger, filter, and sample rate.

If you have a config, you can easily pull those from it:

```
from monkeytype.tracing import CallTracer
from my_mt_config import my_config

logger = my_config.trace_logger()
tracer = CallTracer(
    logger=logger,
    code_filter=my_config.code_filter(),
    sample_rate=my_config.sample_rate(),
)
```

The *CallTracer* has no public API apart from its constructor, but it is suitable for passing to `sys.setprofile` as a profiler:

```
sys.setprofile(tracer)

# run some code to be traced

sys.setprofile(None)  # remove the tracer
```

If your *CallTraceLogger* requires flushing, you should also do this after completing tracing:

```
logger.flush()
```

6.2.4 Deciding which calls to trace

You probably don't want to store traces for every single function called by your program; that will likely include a lot of calls to Python standard-library or third-party library functions that aren't your target for type annotation.

To filter the calls that will be traced, you can return a predicate function from the *code_filter()* method of your *Config()*. This function should take a Python code object and return a boolean: `True` means the function will be traced, and `False` means it will not.

The *DefaultConfig* includes a default code filter. If the environment variable `MONKEYTYPE_TRACE_MODULES` is set to a list of package and/or module names, the default filter traces only code from within those modules. Otherwise, the default filter simply excludes code from the Python standard library and site-packages.

6.2.5 Logging traces

A call-trace logger is responsible for accepting *CallTrace* instances one by one as they are generated by the tracing code and doing something with them. It could print them directly to stdout, in the simplest case, or (more likely) hand them off to a *CallTraceStore* for storage and later retrieval.

class monkeytype.tracing.**CallTraceLogger**
Defines the interface that call-trace loggers should implement.

log(*trace: CallTrace*) → None
Accept a single *CallTrace* and do something with it. This method is called every time a new *CallTrace* is generated.

flush() → None
Flush logged call traces. This method is called once on exiting from the *trace()* context manager.

This method doesn't have to be implemented; by default it is a no-op. For very simple trace loggers (e.g. logging to stdout), each trace can be fully handled in *log()* directly as it is received, and no batching or flushing is needed.

CallTraceStoreLogger

class monkeytype.db.base.**CallTraceStoreLogger**(*store: CallTraceStore*)

The typical function of a call-trace logger is just to batch collected traces and then store them in a *CallTraceStore*. This is implemented by *CallTraceStoreLogger*. Its *log()* method just appends the trace to an in-memory list, and its *flush()* method saves all collected traces to the given `store`.

CallTrace

class monkeytype.tracing.**CallTrace**(*func: Callable, arg_types: Dict[str, type], return_type: Optional[type] = None, yield_type: Optional[type] = None*)

Type information for one traced call of one function.

func: Callable
The function that was called.

funcname: str
Fully-qualified name of the function, including module name (e.g. `some.module.some_func` or `some.module.SomeClass.some_method`).

arg_types: Dict[str, type]
Dictionary mapping argument names to types, for this particular traced call.

return_type: Optional[type]
Type returned by this call, or `None` if this call did not return.

yield_type: Optional[type]
Type yielded by this call, or `None` if this call did not yield.

6.3 Generating type annotations

Use the `monkeytype` command-line script to generate and apply stub files based on recorded call traces.

The script must be able to import your code. It automatically adds the current working directory to the Python path, so ensuring that you run `monkeytype` from the root of your code is usually sufficient. Alternatively, you can set the `PYTHONPATH` environment variable.

6.3.1 monkeytype list_modules

Print a list of all modules for which call traces have been stored.

6.3.2 monkeytype stub

Run `monkeytype stub some.module` to generate a stub file for the given module based on call traces queried from the trace store. If the module already has some type annotations, those annotations will be respected and will not be replaced with annotations derived from traced calls.

The generated stub file will be printed to standard output. If you want to save it to a file, redirect the output to a file (e.g. `monkeytype stub some.module > some/module.pyi`).

You can also run e.g. `monkeytype stub some.module:SomeClass` or `monkeytype stub some.module:somefunc` to generate a stub for just one class or function.

MonkeyType must import your code in order to generate annotations for it, so if a module has import side effects, running `monkeytype stub` on the module will trigger those side effects. For "executable" modules, ensure the execution code is protected with `if __name__ == '__main__'` to avoid MonkeyType triggering it.

6.3.3 monkeytype apply

If you prefer inline type annotations, `monkeytype apply some.module` will generate annotations for `some.module` internally (in exactly the same way as `monkeytype stub` would), but rather than printing the annotations in stub syntax, it will apply them directly to the code file, modifying it in-place.

Obviously this is best used when the file is tracked in a version-control system, so you can easily see the changes made by MonkeyType and accept or reject them. MonkeyType annotations are rarely suitable exactly as generated; they are a starting point and usually require some adjustment by someone who understands the code.

6.3.4 Options

Both `monkeytype stub` and `monkeytype apply` accept the following options:

-c <config-path>, **--config** <config-path>
: The location of the *config object* defining your call-trace store and other configuration. The config-path should take the form `some.module:name`, where `name` is the variable in `some.module` containing your config instance.

  Optionally, the value can also include a `()` suffix, and MonkeyType will call/instantiate the imported function/class with no arguments to get the actual config instance.

  The default value is `monkeytype.config:get_default_config()`, which tries the config path `monkeytype_config:CONFIG` and falls back to `monkeytype.config:DefaultConfig()` if there is no `monkeytype_config` module. This allows creating a custom config that will be used by default just by creating `monkeytype_config.py` with a `CONFIG` instance in it.

-l <limit>, **--limit** <limit>
: The maximum number of call traces to query from your call trace store.

  See the *query_limit()* config method.

  Default: 2000

--disable-type-rewriting
: Don't apply your configured *Type rewriters* to the output types.

Additionally, `monkeytype stub` accepts:

--ignore-existing-annotations
: Generate a stub based only on traced calls, ignoring any existing type annotations in the code. (By default, existing annotations in the code take precedence over traced types.) This can be useful for validating the correctness of existing annotations.

  This option is not usable with `monkeytype apply`, since it would frequently result in a conflict when attempting to apply annotations.

--omit-existing-annotations
: Generate a stub that omits any annotations that are already present in the source. (By default, existing annotations in the source are reproduced in the stub.) Because MonkeyType has to replicate existing annotations via runtime introspection, and doesn't have access to the original string, its replication is often imperfect, which would cause spurious conflicts when applying the stub. Omitting these annotations entirely when generating a stub for application has no cost, since they are already present in the source.

This option is implied by `monkeytype apply`, since it minimizes the possibility of a conflict when attempting to apply annotations.

This option is mutually exclusive with `--ignore-existing-annotations`.

--diff
Generate a textual diff between stubs generated by preserving existing annotations and ignoring them. Use this to see how accurately your annotations represent what is seen in production.

6.3.5 monkeytype list-modules

The `monkeytype list-modules` subcommand outputs a list of all modules which have traces present in the trace store. This command respects only the *`--config`* option.

6.3.6 Type rewriters

MonkeyType's built-in type generation is quite simple: it just makes a `Union` of all the types seen in traces for a given argument or return value, and shrinks that `Union` to remove redundancy. All additional type transformations are performed through configured type rewriters.

class monkeytype.typing.**TypeRewriter**
The *`TypeRewriter`* class provides a type-visitor that can be subclassed to easily implement custom type transformations.

Subclasses can implement arbitrary `rewrite_Foo` methods for rewriting a type named `Foo`. *`TypeRewriter`* itself implements only `rewrite_Dict`, `rewrite_List`, `rewrite_Set`, `rewrite_Tuple`, `rewrite_Union` (in addition to the methods listed below). These methods just recursively rewrite all type arguments of the container types.

For example type rewriter implementations, see the source code of the subclasses listed below.

rewrite(*typ: type*) → type
Public entry point to rewrite given type; return rewritten type.

generic_rewrite(*typ: type*) → type
Fallback method when no specific `rewrite_Foo` method is available for a visited type.

class monkeytype.typing.**RemoveEmptyContainers**
Rewrites e.g. `Union[List[Any], List[int]]` to `List[int]`. The former type frequently occurs when a method that takes `List[int]` also sometimes receives the empty list, which will be typed as `List[Any]`.

class monkeytype.typing.**RewriteConfigDict**
Takes a generated type like `Union[Dict[K, V1], Dict[K, V2]]` and rewrites it to `Dict[K, Union[V1, V2]]`.

class monkeytype.typing.**RewriteLargeUnion**(*max_union_len: int = 5*)
Rewrites large unions (by default, more than 5 elements) to simply *Any*, for better readability of functions that aren't well suited to static typing.

class monkeytype.typing.**ChainedRewriter**(*rewriters: Iterable[TypeRewriter]*)
Accepts a list of rewriter instances and applies each in order. Useful for composing rewriters, since the *`type_rewriter`* config method only allows returning a single rewriter.

class monkeytype.typing.**NoOpRewriter**
Does nothing. The default type rewriter in the base *`Config`*.

6.4 Storing call traces

MonkeyType operates in two phases: *call tracing* and *stub generation*. You first run some code under MonkeyType tracing and store the traced calls. You can do this repeatedly, maybe even sampled in production continually so you always have up-to-date traces available. Then whenever you need, you run `monkeytype stub` or `monkeytype apply` to generate annotations based on types from the recorded traces.

In order to do this, MonkeyType needs a backing store for the recorded call traces. By default it will use *SQLiteStore*, which stores traces in a local SQLite database file. But you can write your own *CallTraceStore* subclass to store traces in whatever data store works best for you, and return an instance of your custom store from the *trace_store()* method of your *Config* class.

6.4.1 CallTraceStore interface

The `CallTraceStore` base class defines the interface that all call-trace stores must implement. The *SQLiteStore* subclass provides a useful example implementation of the *CallTraceStore* interface.

class monkeytype.db.base.**CallTraceStore**

classmethod make_store(*connection_string: str*) → CallTraceStore
Create and return an instance of the store, given a connection string.

The format and interpretation of the connection string is entirely up to the store class. Typically it might be e.g. a URI like `mysql://john:pass@localhost:3306/my_db`.

add(*traces: Iterable[CallTrace]*) → None
Store one or more `CallTrace` instances.

Implementations of this method will probably find the *serialize_traces()* function useful.

filter(*module: str*, *qualname_prefix: Optional[str] = None*, *limit: int = 2000*) → List[CallTraceThunk]
Query call traces from the call trace store. The `module` argument should be provided as a dotted Python import path (e.g. `some.module`).

The store should return the most recent `limit` traces available for the given `module` and `qualname`.

The returned *CallTraceThunk* instances can be any object that implements a *to_trace()* zero-argument method returning a `CallTrace` instance. This allows callers of `filter` to handle deserialization errors as desired per-trace.

Most stores will choose to return instances of *CallTraceRow*, which implements a *to_trace()* that deserializes traces from the same JSON format that its *from_trace()* classmethod serializes to.

list_modules() → List[str]
Query all traces in the trace store and return a list of module names for which traces exist in the store.

6.4.2 SQLiteStore

MonkeyType bundles one sample store implementation, which *DefaultConfig* uses as the default store. It stores call traces in a SQLite database in a local file.

class monkeytype.db.sqlite.**SQLiteStore**

classmethod make_store(*connection_string: str*) → SQLiteStore
The `connection_string` argument will be passed straight through to the Python standard library sqlite module.

add(*traces: Iterable[CallTrace]*) → None
Store one or more `CallTrace` instances in the SQLite database, encoded via *CallTraceRow*.

filter(*module: str*, *qualname_prefix: Optional[str] = None*, *limit: int = 2000*) → List[CallTraceRow]
Query up to `limit` call traces from the SQLite database for a given `module` and optional `qualname_prefix`, returning each as a *CallTraceRow* instance.

6.4.3 serialize_traces

monkeytype.encoding.**serialize_traces**(*traces: Iterable[CallTrace]*) → Iterable[CallTraceRow]
Serialize an iterable of *CallTrace* to an iterable of *CallTraceRow* (via *CallTraceRow.from_trace()*). If any trace fails to serialize, the exception is logged and serialization continues.

6.4.4 CallTraceRow

The *CallTraceRow* class implements serialization/deserialization of *CallTrace* instances to/from JSON. See the implementation of *SQLiteStore* for example usage.

It is not required for a custom store to use *CallTraceRow*; a store may choose to implement its own alternative (de)serialization.

class monkeytype.encoding.**CallTraceRow**

classmethod from_trace(*trace: CallTrace*) → CallTraceRow
Serialize a *CallTraceRow* from the given *CallTrace*.

to_trace() → CallTrace
Deserialize and return the *CallTrace* represented by this *CallTraceRow*.

module: str
The module in which the traced function is defined, e.g. `some.module`.

qualname: str
The `__qualname__` of the traced function or method, e.g. `some_func` for a top-level function or `SomeClass.some_method` for a method.

arg_types: str
A JSON-serialized representation of the concrete argument types for a single traced call. See the implementation for details of the format.

return_type: Optional[str]
A JSON-serialized representation of the actual return type of this traced call, or `None` if this call did not return (i.e. yielded instead).

yield_type: Optional[str]
A JSON-serialized representation of the actual yield type for this traced call, or `None` if this call did not yield (i.e. returned instead).

6.4.5 CallTraceThunk

The minimal required interface of the objects returned from *CallTraceStore.filter()*. Most stores will use *CallTraceRow* to satisfy this interface.

class monkeytype.db.base.**CallTraceThunk**

to_trace() → CallTrace
Produce a *CallTrace* instance based on the serialized trace data stored in this thunk.

6.5 Frequently asked questions

6.5.1 I'm using Django 1.11+ and `monkeytype run manage.py test` generates no traces.

Django 1.11 enabled parallel test runs by default. This means your tests run in separate subprocesses; MonkeyType is tracing only the main supervisor process, which runs only Django code (that is excluded from tracing by the default code filter). To get traces, use `--parallel 1` when running your tests under MonkeyType tracing.

6.5.2 I'm using Django, and I get an AppRegistryNotReady exception when I run monkeytype.

You need to use the *cli_context()* config method to call `django.setup()`.

6.5.3 I run tests against my package installed in site-packages, and I get no traces.

The default *code_filter()* excludes all code from the standard library and `site-packages`, on the assumption that it is third-party code that isn't useful to trace. If you want to trace calls to a package in `site-packages`, you can set the environment variable `MONKEYTYPE_TRACE_MODULES` to a comma-separated list of package/module names you want to trace; only these modules will be traced, wherever they are installed. For more advanced behaviors, you can define *your own custom code filter*.

6.5.4 Why did my test coverage measurement stop working?

MonkeyType uses the same sys.setprofile hook that coverage.py uses to measure Python code coverage, so you can't use MonkeyType and coverage measurement together. If you want to run your tests under MonkeyType tracing, disable coverage measurement for that run, and vice versa.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

Symbols

A

C

D

F

G

L

M

N

P

Q

R

S

T