A Python Metaprogramming Primer

Brian Welch

About me

- Find my projects at https://brianwel.ch

Before we begin

- Slides available at:
  - https://slides.brianwel.ch/python-metaprogramming
- Sample projects available at:
  - https://github.com/welchbj/python-metaprogramming-samples
- All code samples assume a CPython 3.8 environment
- Dunder methods
  - __init__ == “dunder init”
  - __new__ == “dunder new”
  - etc.

Agenda

- High-level overview
- Metaclasses and related constructs
- Hooking the Python `import` system
- Using the `inspect` module
- Applications to a new framework

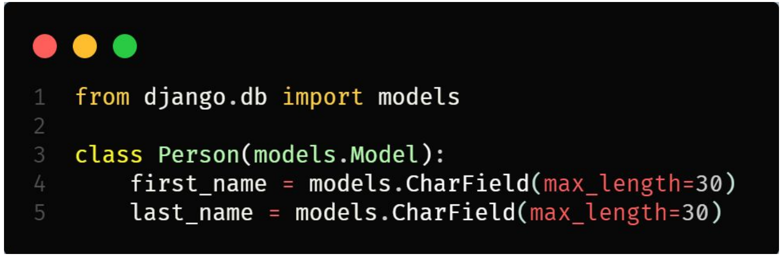
Overview

- What is metaprogramming?
    - Programming with first-class access to language constructs at runtime (or compile-time)
- Why use it?
    - Optimize the developer interface
    - Reduce boilerplate
    - Introspect and mutate Python object definitions, functions, and internals at runtime

Metaclasses and friends

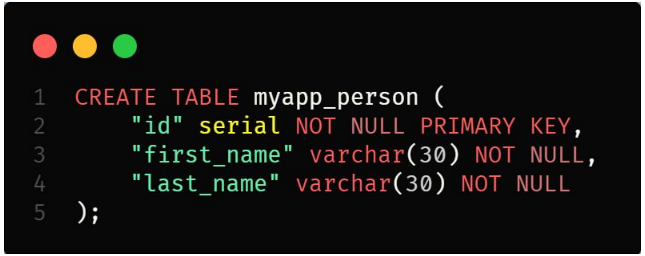
Where have you seen metaclasses?

- ORM model classes
  - Django
  - SQLAlchemy
- All around the standard library
  - Abstract classes in `abc`
  - ASN1 abstractions in `ssl`
  - Node types in `ast`
  - And lots of other modules



```python
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

Sample Django model class from the docs

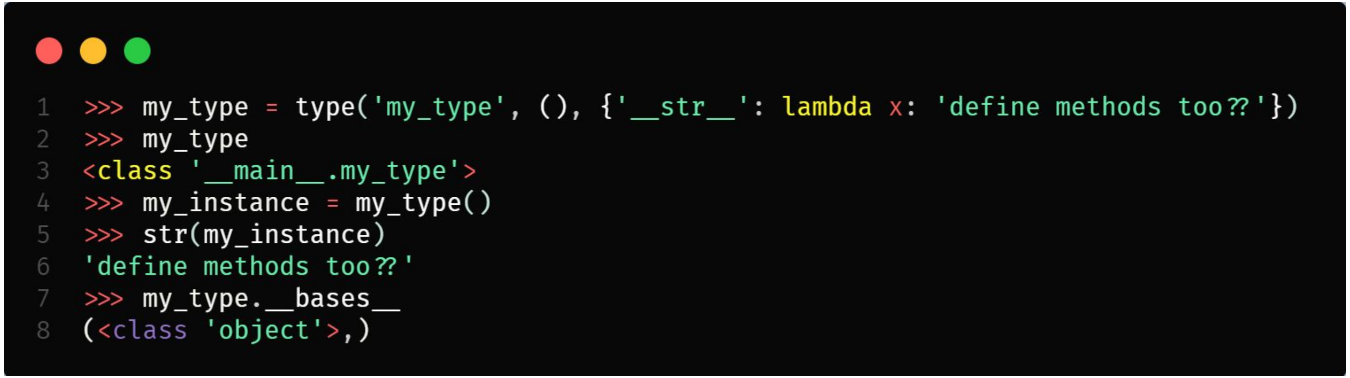


```sql
CREATE TABLE myapp_person (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(30) NOT NULL
);
```

Generated SQL code from the Django model class

How it works

- Think of metaclasses as class factories
- Creating our own metaclass lets us:
  - Hook into the class-creating assembly line
  - Mutate the data model behind typical Python classes
  - Constrain how other developers extend our class definitions
- The most meta of all metaclasses: `type(name, bases, dict)`



```
>>> my_type = type('my_type', (), {'__str__': lambda x: 'define methods too⁇'})
>>> my_type
<class '__main__.my_type'>
>>> my_instance = my_type()
>>> str(my_instance)
'define methods too⁇'
>>> my_type.__bases__
(<class 'object'>,)
```

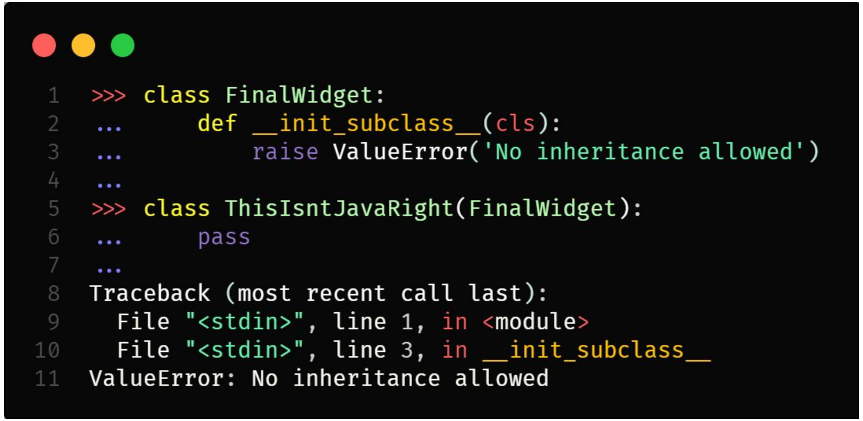
Example dynamic class creation with `type`

`__prepare__`, `__new__`, and `__init__`

- `__prepare__`
  - Alter the class namespace passed to `__new__`
  - Use case: the standard library's `EnumMeta` class
- `__new__`
  - Alter the underlying data model for the instance passed to `__init__`
  - Use case: the standard library's `timedelta` class
- `__init__`
  - What you already know and love for initializing classes

The easy way: `__init_subclass__`

- PEP 487 -- Simpler customisation of class creation (released in Python 3.6)
- Simple solution for common metaprogramming patterns like:
    - Injecting class-level attributes
    - Keeping track of subclasses for plugin-like systems (see: `simple-plugin-system`)
    - Enforcing inheritance constraints (like Java's `final` attribute)



```
>>> class FinalWidget:
...     def __init_subclass__(cls):
...         raise ValueError('No inheritance allowed')
...
>>> class ThisIsntJavaRight(FinalWidget):
...     pass
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in __init_subclass__
ValueError: No inheritance allowed
```

Example `__init_subclass__` *hooking to prevent subclassing*

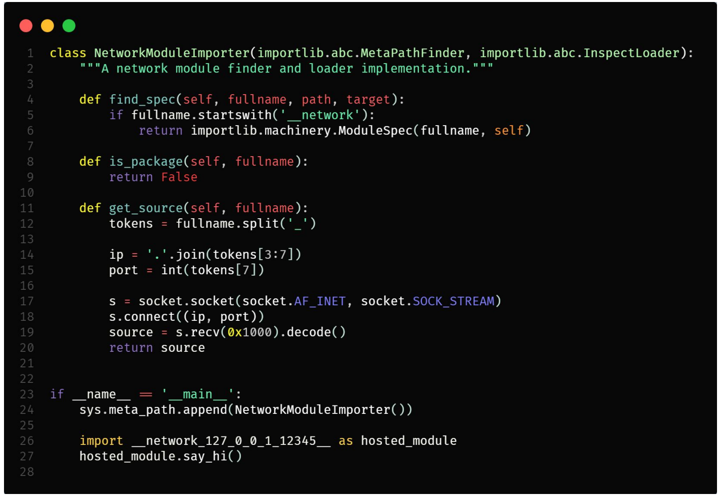
Hooking the `import` flow

`import` machinery

- `sys.meta_path`
  - A list of finders searched when `sys.modules` cannot service an import
  - We can modify this list at any time
- Finders
  - Tell the import engine if a module can be loaded
  - ABCs to build off of: `MetaPathFinder`, `PathEntryFinder`
- Loaders
  - Fetch the code from wherever it is
  - ABCs to build off of: `ResourceLoader`, `InspectLoader`, `FileLoader`

Some potential use cases

- Should you ever do this?
  - Probably not, but there are niche valid use cases
- Load modules from zipfiles via `zipimport`
  - Take it further: embed compressed third-party modules in a portable Python binary
- Load modules over the network
  - See: `network-import-loader`



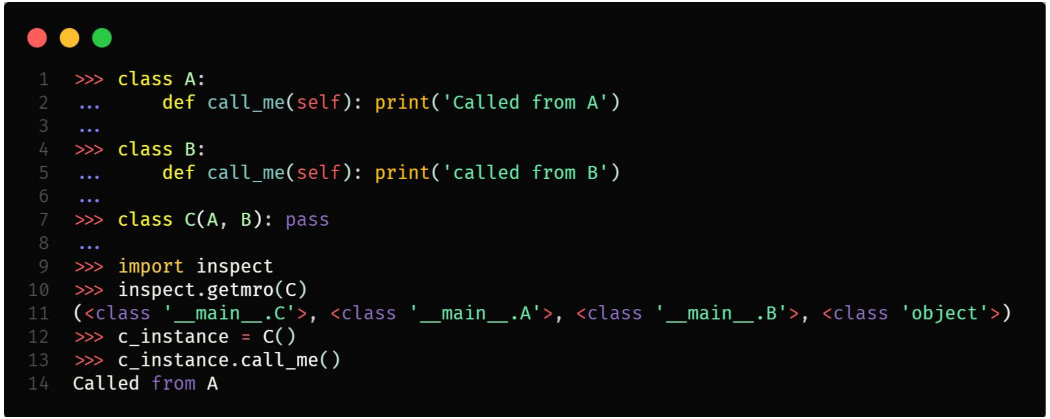
```python
class NetworkModuleImporter(importlib.abc.MetaPathFinder, importlib.abc.InspectLoader):
    """A network module finder and loader implementation."""

    def find_spec(self, fullname, path, target):
        if fullname.startswith('__network'):
            return importlib.machinery.ModuleSpec(fullname, self)

    def is_package(self, fullname):
        return False

    def get_source(self, fullname):
        tokens = fullname.split('_')

        ip = '.'.join(tokens[3:7])
        port = int(tokens[7])

        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((ip, port))
        source = s.recv(0x1000).decode()
        return source


if __name__ == '__main__':
    sys.meta_path.append(NetworkModuleImporter())

    import __network_127_0_0_1_12345__ as hosted_module
    hosted_module.say_hi()
```

A component of a proof-of-concept network import system

The `inspect` Swiss Army Knife

Method Resolution Order (MRO)

- Mitigating the diamond inheritance problem
- Why would we need to use this?
    - Determine the relative “distance” between two class definitions
    - Debugging complicated inheritance models
    - Common exploitation technique in Server-Side Template Injection payloads

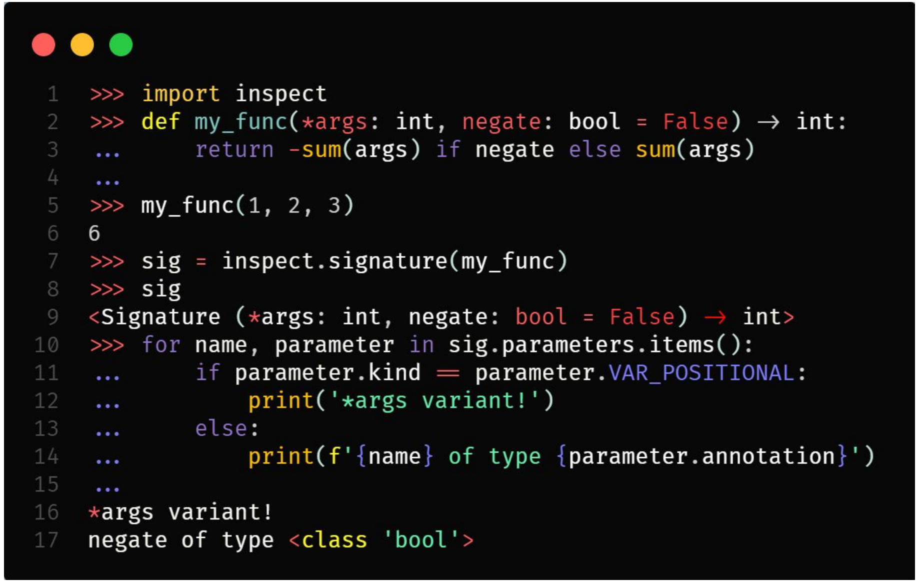


```
>>> class A:
...     def call_me(self): print('Called from A')
...
>>> class B:
...     def call_me(self): print('called from B')
...
>>> class C(A, B): pass
...
>>> import inspect
>>> inspect.getmro(C)
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>)
>>> c_instance = C()
>>> c_instance.call_me()
Called from A
```

Example inspection of a class’s MRO via the `inspect` module

Examining function signatures

- Iterate over function parameters names, types, default values, and more
- Lets us properly handle `*args` and `**kwargs` argument variants

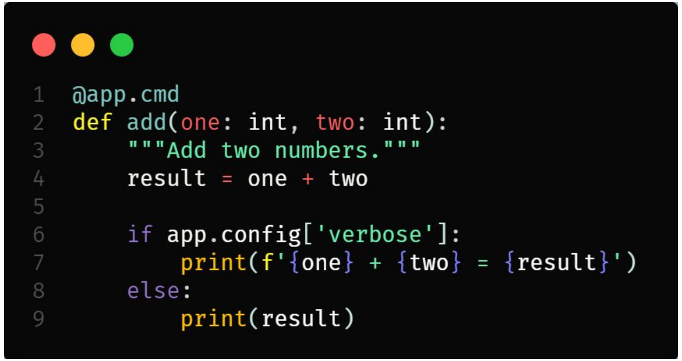


```
>>> import inspect
>>> def my_func(*args: int, negate: bool = False) -> int:
...     return -sum(args) if negate else sum(args)
...
>>> my_func(1, 2, 3)
6
>>> sig = inspect.signature(my_func)
>>> sig
<Signature (*args: int, negate: bool = False) -> int>
>>> for name, parameter in sig.parameters.items():
...     if parameter.kind == parameter.VAR_POSITIONAL:
...         print('*args variant!')
...     else:
...         print(f'{name} of type {parameter.annotation}')
...
*args variant!
negate of type <class 'bool'>
```

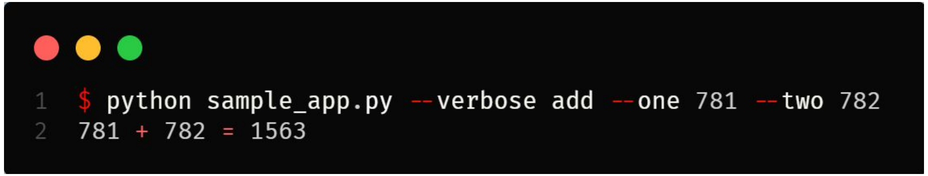
Basic introspection of a function signature with `inspect`

Example application

- Abstracting away the "`argparse` translation layer"
  - Manually writing `ArgumentParser` definitions
  - Boilerplate code to map `argparse` results to your business logic
  - Instead: Derive command-line options directly from Python function signatures
- A simple example: `auto-argparse-with-inspect`
  - A more robust solution: the Click library



```python
@app.cmd
def add(one: int, two: int):
    """Add two numbers."""
    result = one + two

    if app.config['verbose']:
        print(f'{one} + {two} = {result}')
    else:
        print(result)
```

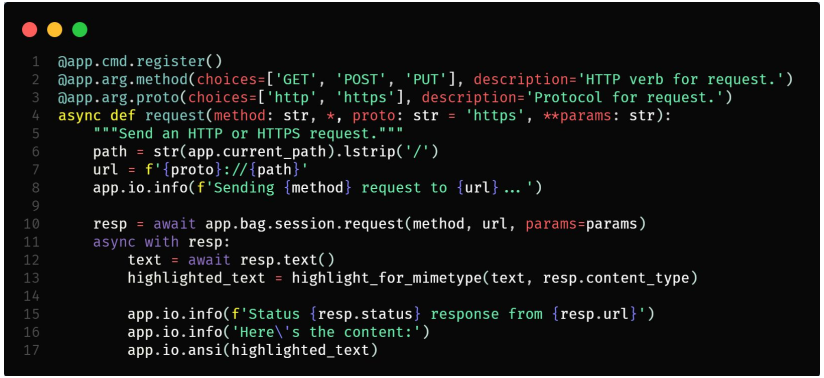


```
$ python sample_app.py --verbose add --one 781 --two 782
781 + 782 = 1563
```

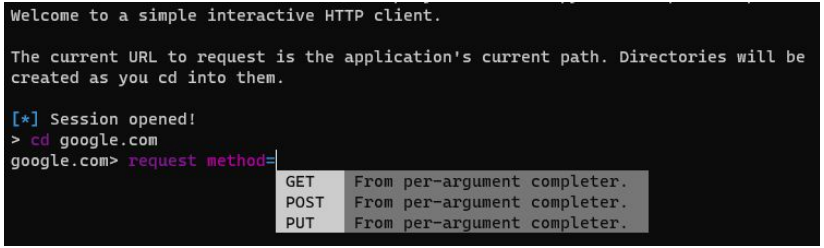
(Arguably) practical applications

The `almanac` framework

- Metaprogramming layer built on some existing libraries
  - Python Prompt Toolkit, Pygments, and pyparsing
- Bind Python functions directly to auto-completed commands in an interactive shell
- Pseudo-filesystem for managing application state



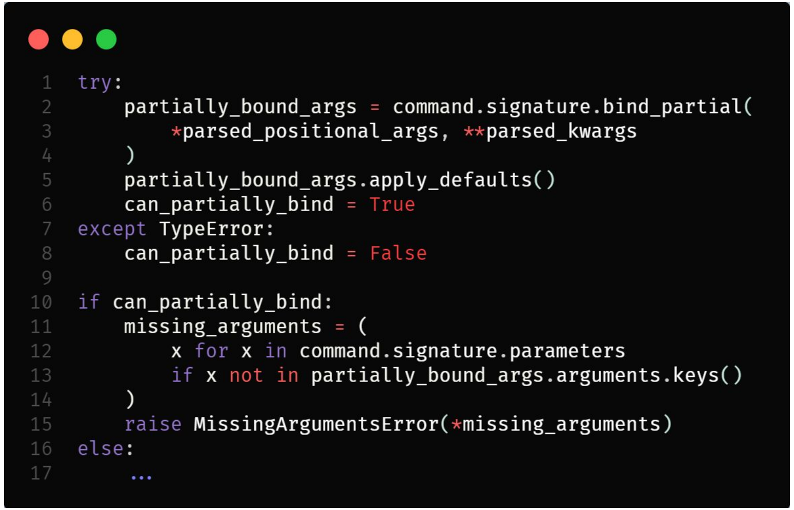
```python
@app.cmd.register()
@app.arg.method(choices=['GET', 'POST', 'PUT'], description='HTTP verb for request.')
@app.arg.proto(choices=['http', 'https'], description='Protocol for request.')
async def request(method: str, *, proto: str = 'https', **params: str):
    """Send an HTTP or HTTPS request."""
    path = str(app.current_path).lstrip('/')
    url = f'{proto}://{path}'
    app.io.info(f'Sending {method} request to {url}... ')

    resp = await app.bag.session.request(method, url, params=params)
    async with resp:
        text = await resp.text()
        highlighted_text = highlight_for_mimetype(text, resp.content_type)

        app.io.info(f'Status {resp.status} response from {resp.url}')
        app.io.info('Here\'s the content:')
        app.io.ansi(highlighted_text)
```



```
Welcome to a simple interactive HTTP client.

The current URL to request is the application's current path. Directories will be
created as you cd into them.

[*] Session opened!
> cd google.com
google.com> request method=
                           GET    From per-argument completer.
                           POST   From per-argument completer.
                           PUT    From per-argument completer.
```

Binding arguments to signatures

- We already know `inspect` lets us introspect function signatures
- We can also try fully and partially applying sets of arguments to signatures
  - See: inspect.Signature.bind_partial
  - Lets us see if a user is missing arguments for a command

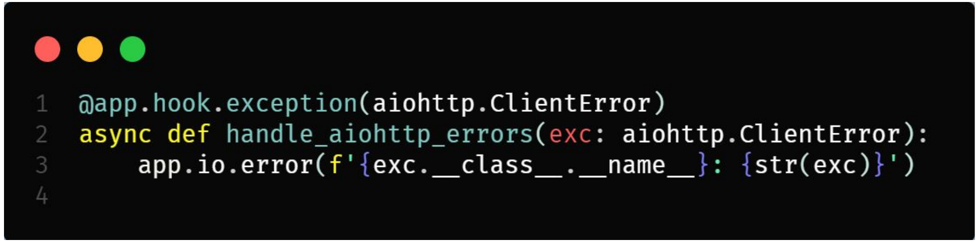


```python
try:
    partially_bound_args = command.signature.bind_partial(
        *parsed_positional_args, **parsed_kwargs
    )
    partially_bound_args.apply_defaults()
    can_partially_bind = True
except TypeError:
    can_partially_bind = False

if can_partially_bind:
    missing_arguments = (
        x for x in command.signature.parameters
        if x not in partially_bound_args.arguments.keys()
    )
    raise MissingArgumentsError(*missing_arguments)
else:
    ...
```

Example code for tring to diagnose invalid arguments for a function signature

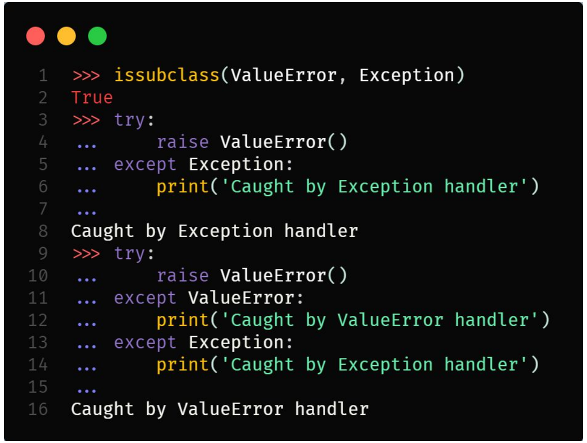
Finding the “closest” `Exception` type

- Exception types can be ambiguous
  - They all extend `Exception`
- If we want to hook a raised exception:
  - We may have multiple “matching” handlers
  - We expect the “closest” handler (i.e., most relevant) to be executed



```
@app.hook.exception(aiohttp.ClientError)
async def handle_aiohttp_errors(exc: aiohttp.ClientError):
    app.io.error(f'{exc.__class__.__name__}: {str(exc)}')

```

Example of hooking an exception type in `almanac`

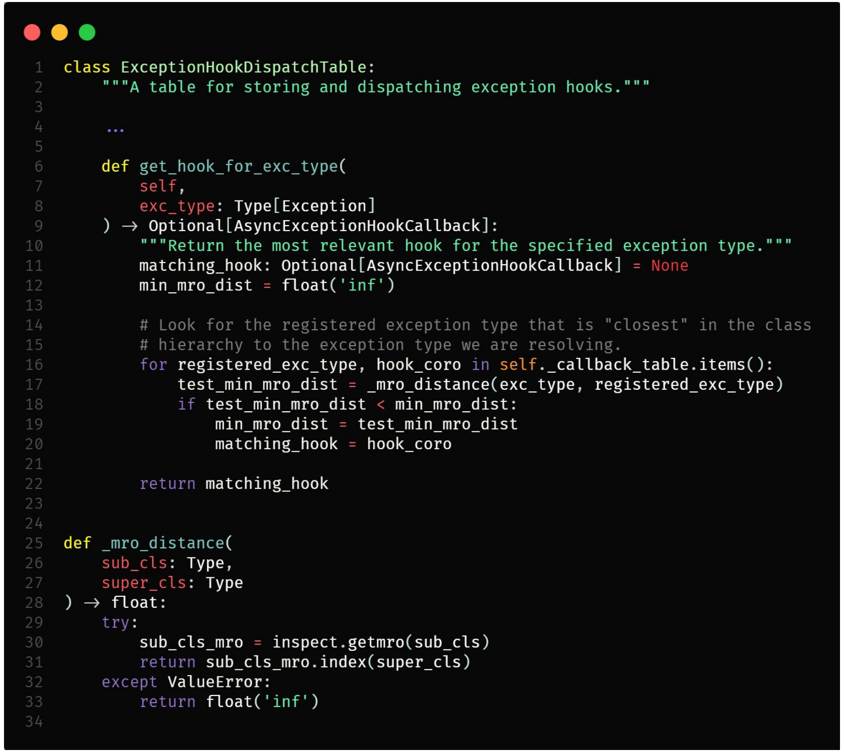


```
>>> issubclass(ValueError, Exception)
True
>>> try:
...     raise ValueError()
... except Exception:
...     print('Caught by Exception handler')
...
Caught by Exception handler
>>> try:
...     raise ValueError()
... except ValueError:
...     print('Caught by ValueError handler')
... except Exception:
...     print('Caught by Exception handler')
...
Caught by ValueError handler
```

Example of potential ambiguities in exception handling

Finding the “closest” `Exception` type (cont.)

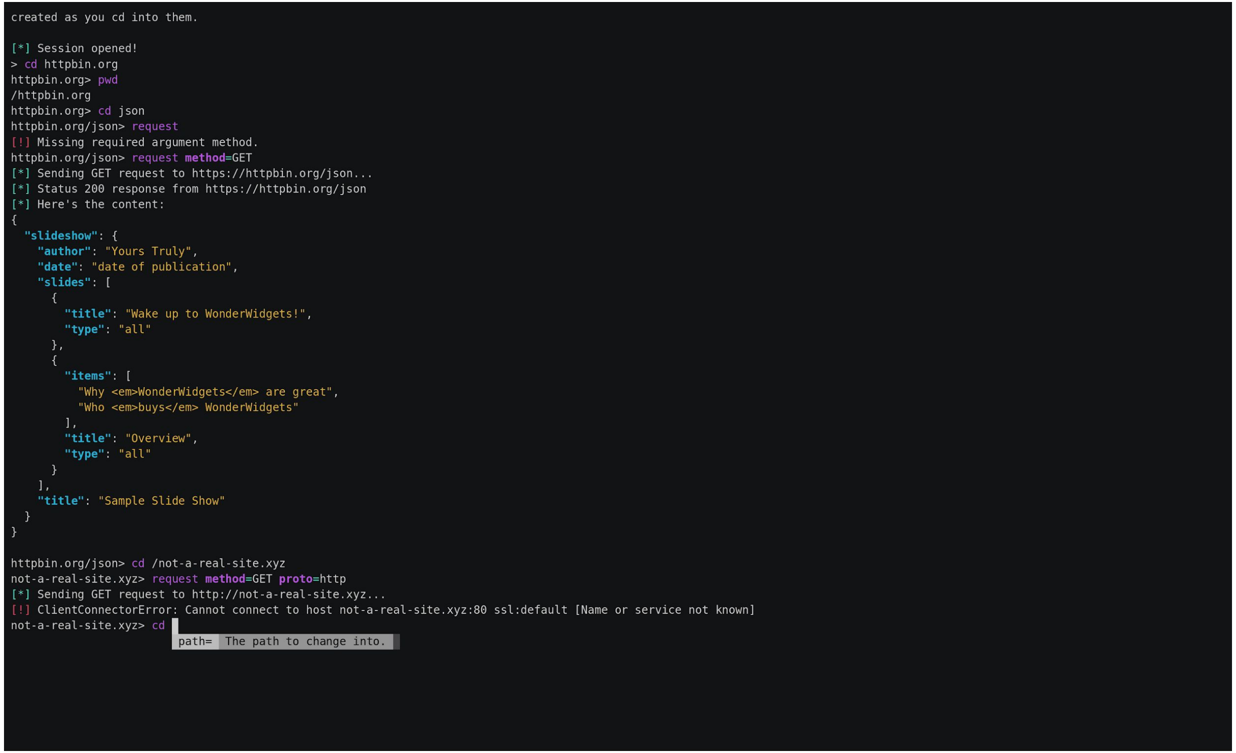
- `inspect` to the rescue!
- Use the concept of “MRO distance”
  - The relative distance between exception super- and sub-types via the sub-type MRO



```python
class ExceptionHookDispatchTable:
    """A table for storing and dispatching exception hooks."""

    ...

    def get_hook_for_exc_type(
        self,
        exc_type: Type[Exception]
    ) -> Optional[AsyncExceptionHookCallback]:
        """Return the most relevant hook for the specified exception type."""
        matching_hook: Optional[AsyncExceptionHookCallback] = None
        min_mro_dist = float('inf')

        # Look for the registered exception type that is "closest" in the class
        # hierarchy to the exception type we are resolving.
        for registered_exc_type, hook_coro in self._callback_table.items():
            test_min_mro_dist = _mro_distance(exc_type, registered_exc_type)
            if test_min_mro_dist < min_mro_dist:
                min_mro_dist = test_min_mro_dist
                matching_hook = hook_coro

        return matching_hook


def _mro_distance(
    sub_cls: Type,
    super_cls: Type
) -> float:
    try:
        sub_cls_mro = inspect.getmro(sub_cls)
        return sub_cls_mro.index(super_cls)
    except ValueError:
        return float('inf')
```

Example exception resolution code, used for deriving the “closest” matching handler

Putting it all together...



```
created as you cd into them.

[*] Session opened!
> cd httpbin.org
httpbin.org> pwd
/httpbin.org
httpbin.org> cd json
httpbin.org/json> request
[!] Missing required argument method.
httpbin.org/json> request method=GET
[*] Sending GET request to https://httpbin.org/json...
[*] Status 200 response from https://httpbin.org/json
[*] Here's the content:
{
  "slideshow": {
    "author": "Yours Truly",
    "date": "date of publication",
    "slides": [
      {
        "title": "Wake up to WonderWidgets!",
        "type": "all"
      },
      {
        "items": [
          "Why <em>WonderWidgets</em> are great",
          "Who <em>buys</em> WonderWidgets"
        ],
        "title": "Overview",
        "type": "all"
      }
    ],
    "title": "Sample Slide Show"
  }
}

httpbin.org/json> cd /not-a-real-site.xyz
not-a-real-site.xyz> request method=GET proto=http
[*] Sending GET request to http://not-a-real-site.xyz...
[!] ClientConnectorError: Cannot connect to host not-a-real-site.xyz:80 ssl:default [Name or service not known]
not-a-real-site.xyz> cd
                         path=  The path to change into.
```

Demo from ~75 lines of `almanac`-powered code

Takeaways

- Build frameworks so you only have to write the boring stuff once
- Balance hidden complexity and developer productivity
- Understand the layer below the one you’re operating at

Thanks for your time

Find these slides at
https://slides.brianwel.ch/python-metaprogramming

Find sample projects at
https://github.com/welchbj/python-metaprogramming-samples

Further resources

- Python internals
  - How Python was Shaped by Leaky Internals, Armin Ronacher (Flask framework)
  - Python Developer Guide: Exploring CPython's Internals
  - Understanding Python Metaclasses
- Import hooking
  - Dependency Injection with Import Hooks in Python 3