Multiple inheritance and super in Python

Dec 10, 2019 13 minutes read #python #mentoring #mozilla

I regularly run mentoring sessions at work in which my mentee Benny and I pick a topic, jump on a video-call for 1-2 hours, share a screen and code on an example project, go over reference documentation and discuss blog posts and conference talks covering the topic. We collect topic ideas for sessions in our 1:1 meeting notes. 📝

During our sessions, Benny and I talk about what I've learned from solving complex problems related to the topic with my team and I share what I found difficult to wrap my head around when I got started. We also discuss next steps, additional resources and any questions Benny might have.

Benny suggested to do a session on object-oriented programming in Python, linear and multiple inheritance and the built-in super function. We also invited Bea, who recently joined the Firefox Telemetry team to the session.

Resources

We all learned a few things about Python from this session and thought it would be cool if we would share our code to help others with their learning journey! 😀

During this mentoring session I recommended Raymond Hettinger's talk "Super considered super!" from PyCon US 2015. I found it useful for learning about cooperative multiple inheritance in Python and it gave me an idea for an example for dependency injection. 📺

I published our code on GitHub under the Mozilla Public License Version 2.0. The repository contains five Python scripts, which build on each other. You will need Python 3.6 or newer and wrapt for example 05. Below you will find the code examples along with my explanation of what I wanted to teach with each of them.

Code example 01

We define a new class `Person` with an instance attribute `name`. It has three methods

that print different text, as well as a custom `__str__` method, which is used when we print the class instance in the other methods. See the Python documentation for more information about `__str__` . 📝

We also write code that is executed when we run the script: It creates a new instance of the `Person` class and calls its methods:

```
# This Source Code Form is subject to the terms of the Mozilla Public
# License, v. 2.0. If a copy of the MPL was not distributed with this
# file, You can obtain one at http://mozilla.org/MPL/2.0/.

"""01 - Define a class in Python."""


class Person:
    """A person has a name and likes to do things."""

    def __init__(self, name: str):
        self.name = name

    def __str__(self) -> str:
        return f"{self.name}"

    def go_to_the_movies(self) -> None:
        print(f"{self} goes to the movies. 🍿")

    def go_hiking(self) -> None:
        print(f"{self} goes hiking. ⛰")

    def build_a_robot(self) -> None:
        print(f"{self} builds a robot. 🤖")


if __name__ == "__main__":
    simone = Person("Simone")
    simone.go_to_the_movies()
    simone.build_a_robot()
    simone.go_hiking()
```

```
$ python examples/01_defining_classes.py
Simone goes to the movies. 🍿
Simone builds a robot. 🤖
```

```
Simone goes hiking. ▲
```

You can find this example on GitHub.

Code example 02

In the next example, we use inheritance to extend the `Person` class. We define a subclass `TeamMember`, which implements a `work_on_project()` method. This method accepts an instance of another new class `Project`.

Inheritance allows us to re-use the code from `Person` in `TeamMember`.

```
# This Source Code Form is subject to the terms of the Mozilla Public
# License, v. 2.0. If a copy of the MPL was not distributed with this
# file, You can obtain one at http://mozilla.org/MPL/2.0/.

"""02 - Extending a class in Python.

Use inheritance to extend the Person class.
"""


class Person:
    """A person has a name and likes to do things."""

    def __init__(self, name: str):
        self.name = name

    def __str__(self) -> str:
        return f"{self.name}"

    def go_to_the_movies(self) -> None:
        print(f"{self} goes to the movies. 🏠")

    def go_hiking(self) -> None:
        print(f"{self} goes hiking. ▲")

    def build_a_robot(self) -> None:
        print(f"{self} builds a robot. 🤖")


class Project:
    """A project has a board name and a description."""
```

```
        A project has a board_name and a description.

    def __init__(self, board_name: str, description: str):
        self.board_name = board_name
        self.description = description

    def __str__(self) -> str:

        return f"Project '{self.board_name}'"


class TeamMember(Person):
    """A team member is a person, who works on projects."""

    def work_on_project(self, project: Project) -> None:
        print(f"{self} is now working on {project}. 📋")


if __name__ == "__main__":
    simone = TeamMember("Simone")
    simone.go_to_the_movies()
    simone.build_a_robot()
    simone.go_hiking()

    data_platform = Project(
        board_name="Data Platform",
        description="Platform providing datasets and data viewing tools.",
    )
    simone.work_on_project(data_platform)
```

```
$ python examples/02_extending_classes.py
Simone goes to the movies. 🍿
Simone builds a robot. 🤖
Simone goes hiking. ⛰
Simone is now working on Project 'Data Platform'. 📋
```

You can find this example on GitHub.

Code example 03

Python also supports class attributes which are shared between all of its instances. We

add an `expertise` class attribute to the `TeamMember` class and use it in the `__str__` method when generating the string representation for the instance. We add several subclasses of `TeamMember` that all override `expertise` with a different value.

This demonstrates that a subclass can override attributes, which are used by its base classes. We don't need to modify `__str__` of `TeamMember` to make this work. That's super powerful! 🦸

Additionally we add a `stay_hydrated()` method to `Person` which will automatically be available in all subclasses and we call that in `work_on_project()` of `TeamMember`.

```
# This Source Code Form is subject to the terms of the Mozilla Public
# License, v. 2.0. If a copy of the MPL was not distributed with this
# file, You can obtain one at http://mozilla.org/MPL/2.0/.

"""03 - Overriding class attributes in Python.

Use inheritance to override a class attribute of the Person class and call
method of base class.
"""

import typing


class Person:
    """A person has a name and likes to do things."""

    def __init__(self, name: str):
        self.name = name

    def __str__(self) -> str:
        return f"{self.name}"

    def stay_hydrated(self) -> None:
        print(f"{self} drinks some water. 🚰")

    def go_to_the_movies(self) -> None:
        print(f"{self} goes to the movies. 🍿")

    def go_hiking(self) -> None:
        print(f"{self} goes hiking. ⛰")

    def build_a_robot(self) -> None:
```

```
        print(f"{self} builds a robot. 🤖")


class Project:
    """A project has a board_name and a description."""


    def __init__(self, board_name: str, description: str):
        self.board_name = board_name
        self.description = description

    def __str__(self) -> str:
        return f"Project '{self.board_name}'"


class TeamMember(Person):
    """A team member is a person, who works on projects, and may have
    specialized in a specific field.
    """

    expertise: typing.Optional[str] = None

    def __str__(self) -> str:
        # Get default string representation from the super class
        default = super().__str__()

        if self.expertise is None:
            return f"{default}"

        return f"{self.expertise} {default}"

    def work_on_project(self, project: Project) -> None:
        """Start working on the given project."""
        print(f"{self} is now working on {project}. 📋")
        self.stay_hydrated()


class MobileEngineer(TeamMember):
    """Team member specialized in developing for mobile platforms."""

    expertise = "📱"


class DataScientist(TeamMember):
```

```
    """Team member specialized in data science."""

    expertise = "📈"


class ProjectManager(TeamMember):
    """Team member specialized in project management."""

    expertise = "📝"


class OperationsEngineer(TeamMember):
    """Team member specialized in running cloud infrastructure."""

    expertise = "📄"


if __name__ == "__main__":
    simone = OperationsEngineer("Simone")
    simone.go_to_the_movies()
    simone.build_a_robot()
    simone.go_hiking()

    chelsea = DataScientist("Chelsea")
    dave = ProjectManager("Dave")
    marlene = MobileEngineer("Marlene")

    data_platform = Project(
        board_name="Data Platform",
        description="Platform providing datasets and data viewing tools.",
    )
    simone.work_on_project(data_platform)
    chelsea.work_on_project(data_platform)
    dave.work_on_project(data_platform)
    marlene.work_on_project(data_platform)
```

```
$ python examples/03_overriding_class_attributes.py
📄 Simone goes to the movies. 🏠
📄 Simone builds a robot. 🤖
📄 Simone goes hiking. ⛰️
📄 Simone is now working on Project 'Data Platform'. 📋
📄 Simone drinks some water. 🚰
```

```
Chelsea is now working on Project 'Data Platform'.
Chelsea drinks some water.
Dave is now working on Project 'Data Platform'.
Dave drinks some water.
Marlene is now working on Project 'Data Platform'.
Marlene drinks some water.
```

You can find this example on GitHub.

Code example 04

The next example introduces a number of changes. Don't feel bad if you don't understand the code right away! I suggest you clone the repository and run the script yourself, comment out code and see what happens. If you have questions beyond that, please feel free to create a new issue on GitHub or ask me on Twitter. ☺

What if a `Person` prefers to drink tea rather than water? Can we override the `stay_hydrated()` method without creating a copy of `Person`, `TeamMember` and all of its subclasses? The answer is yes: We can use multiple inheritance for that!

First we create a new subclass from `Person` named `TeaPerson` and override the `stay_hydrated()` method:

```
class TeaPerson(Person):
    """A person who prefers tea over water."""

    def stay_hydrated(self) -> None:
        print(f"{self} drinks tea. 🍵")
```

Then, if we want to create a new `Person` subclass for people who'd rather drink tea, we add `TeaPerson` to the list of base classes:

```
class DataScientistWhoLikesTea(DataScientist, TeaPerson):
    """Data scientist who prefers tea over water."""
```

When Python looks for a `stay_hydrated()` method it will find the method on `TeaPerson` before it gets to the method on `Person`. If you're curious to find out more about how exactly this works, check out the Python documentation on the Method Resolution Order. 📝

We use the same technique for a slightly more complex class, `RemoteTeamMember`, in this

We use the same technique for a slightly more complex class `RemoteTeamMember` in this example. We add a new contextmanager method `commute()` to `TeamMember` and override it on `RemoteTeamMember`. We then update `work_on_project()` to run its code inside of the context from `commute()`. This will hopefully make sense when you read the code and output below.

What's important is that we can now inject a version of `commute` to subclasses of `TeamMember` by creating new subclasses with an additional base class.

```
class ProjectManagerWhoWorksRemotely(ProjectManager, RemoteTeamMember):
    """Project manager who works remotely."""
```

We can even create a class for a `TeamMember` specialized in developing for mobile platforms, who likes tea and works remotely! 🏠

```
class MobileEngineerWhoWorksRemotelyAndLikesTea(
    MobileEngineer, RemoteTeamMember, TeaPerson
):
    """Mobile engineer who works remotely and prefers tea over water."""
```

Please note that in order to make cooperative multiple inheritance work in Python, we need our classes to implement `__init__` methods that accept required arguments as keyword arguments and forward any additional arguments to their respective super class as determined by Python's Method Resolution Order. ⚠

Here's the complete code example:

```
# This Source Code Form is subject to the terms of the Mozilla Public
# License, v. 2.0. If a copy of the MPL was not distributed with this
# file, You can obtain one at http://mozilla.org/MPL/2.0/.

"""04 - Inject functionality into a class in Python.

Use multiple inheritance to override functionality on base classes.
"""

import contextlib
import typing
```

```python
class Person:
    """A person has a name and likes to do things."""

    def __init__(self, *, name: str, **kwargs: typing.Any):
        self.name = name


    def __str__(self) -> str:
        return f"{self.name}"

    def stay_hydrated(self) -> None:
        print(f"{self} drinks some water. 🚰")

    def go_to_the_movies(self) -> None:
        print(f"{self} goes to the movies. 🎦")

    def go_hiking(self) -> None:
        print(f"{self} goes hiking. ⛰️")

    def build_a_robot(self) -> None:
        print(f"{self} builds a robot. 🤖")


class TeaPerson(Person):
    """A person who prefers tea over water."""

    def stay_hydrated(self) -> None:
        print(f"{self} drinks tea. 🍵")


class Project:
    """A project has a board_name and a description."""

    def __init__(self, board_name: str, description: str):
        self.board_name = board_name
        self.description = description

    def __str__(self) -> str:
        return f"Project '{self.board_name}'"


class TeamMember(Person):
    """A team member is a person, who works on projects, and may have
    specialized in a specific field.
```

```
    """

    expertise: typing.Optional[str] = None

    def __str__(self) -> str:
        # Get default string representation from the super class
        default = super().__str__()

        if self.expertise is None:
            return f"{default}"

        return f"{self.expertise} {default}"

    @contextlib.contextmanager
    def commute(self) -> typing.Generator:
        """Commute to the office and back."""
        print(f"{self} commutes to the office. 🏢")
        yield
        print(f"{self} commutes home. 🏠")

    def work_on_project(self, project) -> None:
        """Start working on the given project."""
        with self.commute():
            print(f"{self} is now working on {project}. 📋")
            self.stay_hydrated()


class MobileEngineer(TeamMember):
    """Team member specialized in developing for mobile platforms."""

    expertise = "📱"


class DataScientist(TeamMember):
    """Team member specialized in data science."""

    expertise = "📈"


class ProjectManager(TeamMember):
    """Team member specialized in project management."""

    expertise = "📝"
```

```python
class OperationsEngineer(TeamMember):
    """Team member specialized in running cloud infrastructure."""

    expertise = "  "


class RemoteTeamMember(TeamMember):
    """Team member who works remotely."""

    def __init__(self, *, workplace: str, **kwargs: typing.Any):
        self.workplace = workplace
        # Forward kwargs to super class
        super().__init__(**kwargs)

    @contextlib.contextmanager
    def commute(self) -> typing.Generator:
        """Stay at home or commute to the workplace and back."""

        if self.workplace == "home":
            print(f"{self} works from home.  ")
            yield
            return

        print(f"{self} commutes to {self.workplace}.  ")
        yield
        print(f"{self} commutes home.  ")


class DataScientistWhoLikesTea(DataScientist, TeaPerson):
    """Data scientist who prefers tea over water."""


class ProjectManagerWhoWorksRemotely(ProjectManager, RemoteTeamMember):
    """Project manager who works remotely."""


class MobileEngineerWhoWorksRemotelyAndLikesTea(
    MobileEngineer, RemoteTeamMember, TeaPerson
):
    """Mobile engineer who works remotely and prefers tea over water."""


if __name__ == "__main__":
```

```
    simone = OperationsEngineer(name="Simone")
    simone.go_to_the_movies()
    simone.build_a_robot()
    simone.go_hiking()

    chelsea = DataScientistWhoLikesTea(name="Chelsea")
    dave = ProjectManagerWhoWorksRemotely(
        name="Dave", workplace="a local coffee shop"
    )
    marlene = MobileEngineerWhoWorksRemotelyAndLikesTea(
        name="Marlene", workplace="home"
    )

    data_platform = Project(
        board_name="Data Platform",
        description="Platform providing datasets and data viewing tools.",
    )
    simone.work_on_project(data_platform)
    chelsea.work_on_project(data_platform)
    dave.work_on_project(data_platform)
    marlene.work_on_project(data_platform)
```

```
$ python examples/04_injecting_functionality.py
📄 Simone goes to the movies. 🍿
📄 Simone builds a robot. 🤘
📄 Simone goes hiking. ⛰️
📄 Simone commutes to the office. 🏢
📄 Simone is now working on Project 'Data Platform'. 📋
📄 Simone drinks some water. 🚰
📄 Simone commutes home. 🏠
📈 Chelsea commutes to the office. 🏢
📈 Chelsea is now working on Project 'Data Platform'. 📋
📈 Chelsea drinks tea. 🍵
📈 Chelsea commutes home. 🏠
📝 Dave commutes to a local coffee shop. 🚌
📝 Dave is now working on Project 'Data Platform'. 📋
📝 Dave drinks some water. 🚰
📝 Dave commutes home. 🏠
📱 Marlene works from home. 🏠
📱 Marlene is now working on Project 'Data Platform'. 📋
📱 Marlene drinks tea. 🍵
```

You can find this example on GitHub.

Code example 05

Bea asked in our mentoring session if there's a way to print information on where a method is defined whenever we call a method of a given subclass. This would be quite helpful to visualize how super works. Benny asked if we could use a decorator or a metaclass for that. I think those were fantastic questions! 👏

Together, we managed to implement a decorator that prints out what we wanted, but we needed to add it to every method to get it to work. A metaclass might be the right tool for this problem.

I personally don't find writing metaclasses simple and couldn't remember the arguments for `__new__` when defining the metaclass. The snippets we found online didn't work for us and we were already over time with the session, so I promised I would publish a working example along with the other code examples here:

```
# This Source Code Form is subject to the terms of the Mozilla Public
# License, v. 2.0. If a copy of the MPL was not distributed with this
# file, You can obtain one at http://mozilla.org/MPL/2.0/.

"""05 - Debug the Method Resolution Order of Python.

Use a metaclass to decorate methods of Person and its subclasses.
"""

import contextlib
import typing

import wrapt


@wrapt.decorator
def log_call(wrapped_method, instance, args, kwargs) -> typing.Any:
    """Print when the decorated method is called."""
    method_name = wrapped_method.__qualname__
    class_name = instance.__class__.__qualname__
    print(f"Calling method {method_name} for {class_name}.")
    return wrapped_method(*args, **kwargs)
```

```
class LogMethods(type):
    """Metaclass that decorates methods with log_call."""

    def __new__(cls, name, bases, attrs, **kwargs):
        for attr, value in attrs.items():

            if not callable(value):
                continue
            attrs[attr] = log_call(value)
        return super().__new__(cls, name, bases, attrs, **kwargs)
```

We can apply this metaclass to `Person` as follows:

```
class Person(metaclass=LogMethods):
    """A person has a name and likes to do things."""
```

Aside from the metaclass, example 05 is identical to example 04. 💻

Now if we run the example it will print a lot of information. The following is an excerpt from the output, which shows what's happening when we call `work_on_project()` on an instance of the `DataScientistWhoLikesTea` class:

```
Calling method TeamMember.work_on_project for DataScientistWhoLikesTea.
Calling method TeamMember.commute for DataScientistWhoLikesTea.
Calling method TeamMember.__str__ for DataScientistWhoLikesTea.
Calling method Person.__str__ for DataScientistWhoLikesTea.
☑ Chelsea commutes to the office. 🏢
Calling method TeamMember.__str__ for DataScientistWhoLikesTea.
Calling method Person.__str__ for DataScientistWhoLikesTea.
☑ Chelsea is now working on Project 'Data Platform'. 📋
Calling method TeaPerson.stay_hydrated for DataScientistWhoLikesTea.
Calling method TeamMember.__str__ for DataScientistWhoLikesTea.
Calling method Person.__str__ for DataScientistWhoLikesTea.
☑ Chelsea drinks tea. 🍵
Calling method TeamMember.__str__ for DataScientistWhoLikesTea.
Calling method Person.__str__ for DataScientistWhoLikesTea.
☑ Chelsea commutes home. 🏠
```

You can find this example on GitHub.

Conclusion

We developed a series of Python scripts, each building on the previous one, that implement cooperative multiple inheritance using `super()`. We did that to learn how to re-use code and override functionality in subclasses. You can find all of our code examples on GitHub. 💻

Make sure to watch Raymond Hettinger's talk on YouTube if you want to learn more about the Method Resolution Order in Python.

Sincere thanks to my co-workers Bea and Benny for asking great questions and coding with me on this example project. It was great fun! Thanks also to my co-worker Peter for proofreading this blogpost. 📝

Back to posts