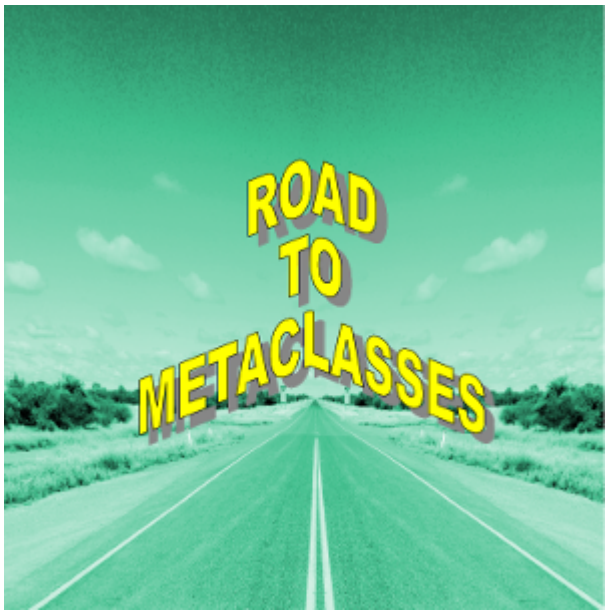
ON THE ROAD TO METACLASSES

MOTIVATION FOR METACLASSES



In this chapter of our tutorial we want to provide some incentives or motivation for the use of metaclasses. To demonstrate some design problems, which can be solved by metaclasses, we will introduce and design a bunch of philosopher classes. Each philosopher class (Philosopher1, Philosopher2, and so on) need the same "set" of methods (in our example just one, i.e. "the_answer") as the basics for his or her pondering and brooding. A stupid way to implement the classes consists in having the same code in every philospher class:

```
class Philosopher1:

    def the_answer(self, *args):
        return 42

class Philosopher2:
    def the_answer(self, *args):
        return 42

class Philosopher3:
    def the_answer(self, *args):
        return 42

plato = Philosopher1()
print(plato.the_answer())
kant = Philosopher2()
# let's see what Kant has to say :-)
print(kant.the_answer())

42
42
```

We can see that we have multiple copies of the method "the_answer". This is error prone and tedious to maintain, of course.

From what we know so far, the easiest way to accomplish our goal without creating redundant code consists in designing a base, which contains "the_answer" as a method. Each Philosopher class inherits now from this base class:

```
class Answers:
    def the_answer(self, *args):
        return 42

class Philosopher1(Answers):
    pass
class Philosopher2(Answers):
    pass
class Philosopher3(Answers):
    pass
plato = Philosopher1()
print(plato.the_answer())
kant = Philosopher2()
# let's see what Kant has to say :-)
print(kant.the_answer())

42
42
```

The way we have designed our classes, each Philosopher class will always have a method "the_answer". Let's assume, we don't know a priori if we want or need this method. Let's assume that the decision, if the classes have to be augmented, can only be made at runtime. This decision might depend on configuration files, user input or some calculations.

```
# the following variable would be set as the result of a runtime
calculation:
x = input("Do you need the answer? (y/n): ")
if x=="y":
    required = True
else:
    required = False

def the_answer(self, *args):
        return 42

class Philosopher1:
    pass
if required:
    Philosopher1.the_answer = the_answer

class Philosopher2:
    pass
if required:
    Philosopher2.the_answer = the_answer

class Philosopher3:
    pass
```

```
if required:
    Philosopher3.the_answer = the_answer


plato = Philosopher1()
kant = Philosopher2()
# let's see what Plato and Kant have to say :-)
if required:
    print(kant.the_answer())
    print(plato.the_answer())
else:
    print("The silence of the philosphers")

Do you need the answer? (y/n): y
42
42
```

Even though this is another solution to our problem, there are still some serious drawbacks. It's error-prone, because we have to add the same code to every class and it seems likely that we might forget it. Besides this it's getting hardly manageable and maybe even confusing, if there are many methods we want to add.

We can improve our approach by defining a manager function and avoiding redundant code this way. The manager function will be used to augment the classes conditionally.

```
# the following variable would be set as the result of a runtime
calculation:
x = input("Do you need the answer? (y/n): ")
if x=="y":
    required = True
else:
    required = False

def the_answer(self, *args):
        return 42

# manager function
def augment_answer(cls):
    if required:
        cls.the_answer = the_answer


class Philosopher1:
    pass
augment_answer(Philosopher1)
class Philosopher2:
    pass
```

```
augment_answer(Philosopher2)
class Philosopher3:
    pass
augment_answer(Philosopher3)


plato = Philosopher1()
kant = Philosopher2()
# let's see what Plato and Kant have to say :-)
if required:
    print(kant.the_answer())
    print(plato.the_answer())
else:
    print("The silence of the philosphers")

Do you need the answer? (y/n): y
42
42
```

This is again useful to solve our problem, but we, i.e. the class designers, must be careful not to forget to call the manager function "augment_answer". The code should be executed automatically. We need a way to make sure that "some" code might be executed automatically after the end of a class definition.

```
# the following variable would be set as the result of a runtime
calculation:
x = input("Do you need the answer? (y/n): ")
if x=="y":
    required = True
else:
    required = False


def the_answer(self, *args):
        return 42

def augment_answer(cls):
    if required:
        cls.the_answer = the_answer
    # we have to return the class now:
    return cls


@augment_answer
class Philosopher1:
    pass
@augment_answer
class Philosopher2:
```

```
    pass
@augment_answer
class Philosopher3:
    pass


plato = Philosopher1()
kant = Philosopher2()


# let's see what Plato and Kant have to say :-)
if required:
    print(kant.the_answer())
    print(plato.the_answer())
else:
    print("The silence of the philosphers")

Do you need the answer? (y/n): y
42
42
```

Metaclasses can also be used for this purpose as we will learn in the next chapter.