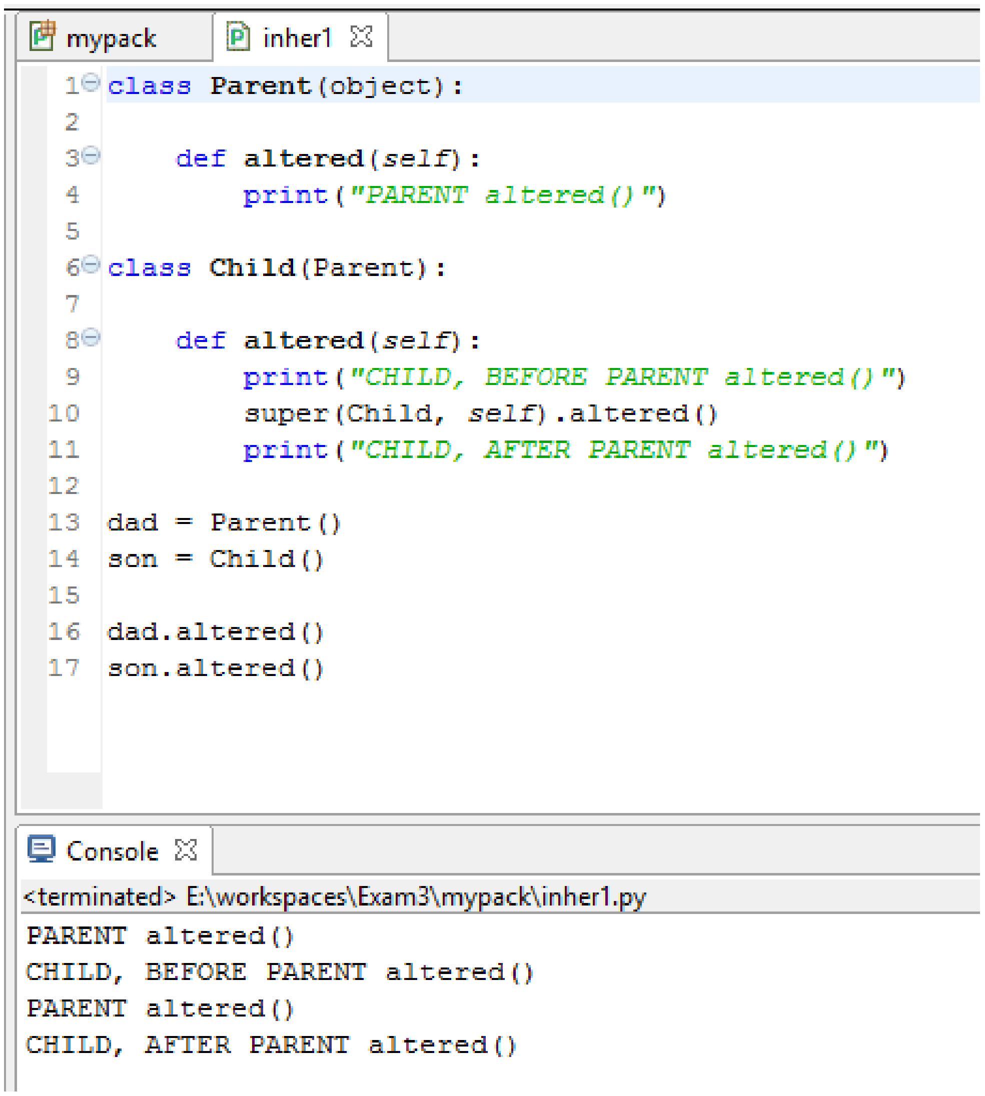
PYTHON

3. INHERITANCE

The way to use inheritance is a special case of overriding where you want to alter the behavior before or after the Parent class's version runs. You first override the function just like in the last example, but then you use a Python built-in function named super to get the Parent version to call. Here's the example of doing that so you can make sense of this description:



```
 1 class Parent(object):
 2 
 3     def altered(self):
 4         print("PARENT altered()")
 5 
 6 class Child(Parent):
 7 
 8     def altered(self):
 9         print("CHILD, BEFORE PARENT altered()")
10         super(Child, self).altered()
11         print("CHILD, AFTER PARENT altered()")
12 
13 dad = Parent()
14 son = Child()
15 
16 dad.altered()
17 son.altered()
```

<terminated> E:\workspaces\Exam3\mypack\inher1.py

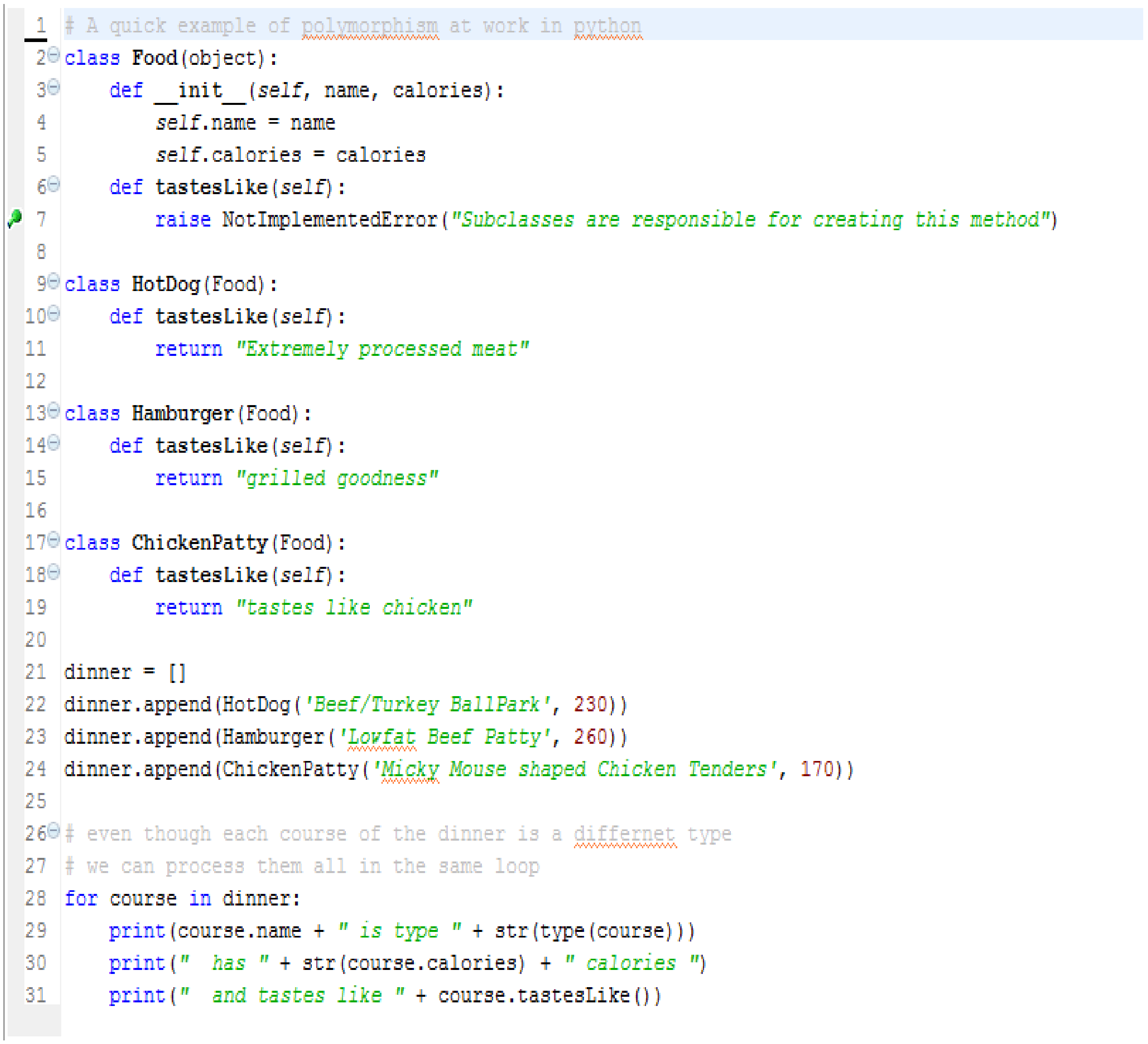
```
PARENT altered()
CHILD, BEFORE PARENT altered()
PARENT altered()
CHILD, AFTER PARENT altered()
```

The important lines here are 9-11, where in the Child I do the following when son.altered() is called:

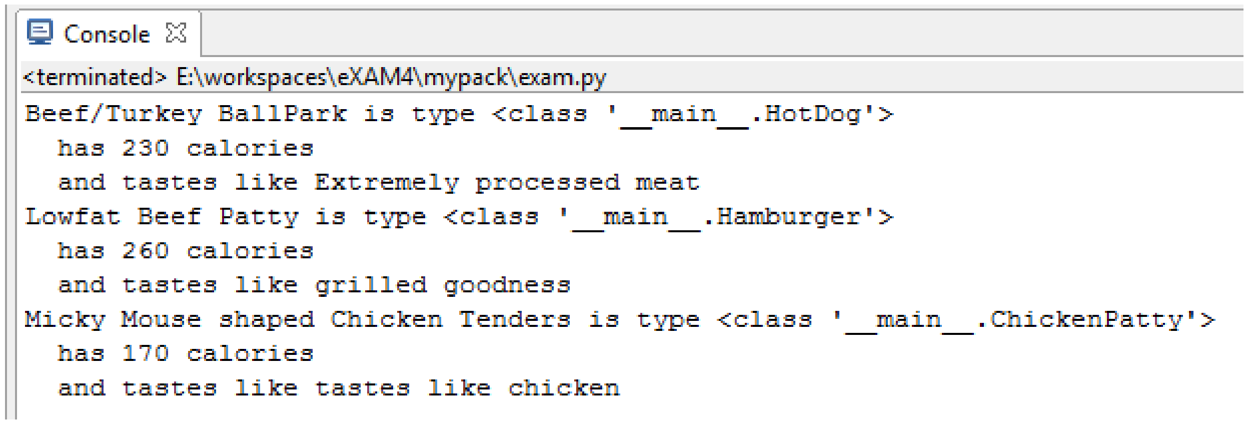
1. Because I've overridden Parent.altered the Child.altered version runs, and line 9 executes like you'd expect.
2. In this case I want to do a before and after so after line 9, I want to use super to get the Parent.alteredversion.
3. On line 10 I call super(Child, self).altered(), which is aware of inheritance and will get the Parent class for you. You should be able to read this as "call super with arguments Child and self, then call the functionaltered on whatever it returns."
4. At this point, the Parent.altered version of the function runs, and that prints out the Parent message.

5. Finally, this returns from the Parent.altered and the Child.altered function continues to print out the after message.

4. POLYMORPHISM



```python
# A quick example of polymorphism at work in python
class Food(object):
    def __init__(self, name, calories):
        self.name = name
        self.calories = calories
    def tastesLike(self):
        raise NotImplementedError("Subclasses are responsible for creating this method")

class HotDog(Food):
    def tastesLike(self):
        return "Extremely processed meat"

class Hamburger(Food):
    def tastesLike(self):
        return "grilled goodness"

class ChickenPatty(Food):
    def tastesLike(self):
        return "tastes like chicken"

dinner = []
dinner.append(HotDog('Beef/Turkey BallPark', 230))
dinner.append(Hamburger('Lowfat Beef Patty', 260))
dinner.append(ChickenPatty('Micky Mouse shaped Chicken Tenders', 170))

# even though each course of the dinner is a differnet type
# we can process them all in the same loop
for course in dinner:
    print(course.name + " is type " + str(type(course)))
    print("  has " + str(course.calories) + " calories ")
    print("  and tastes like " + course.tastesLike())
```

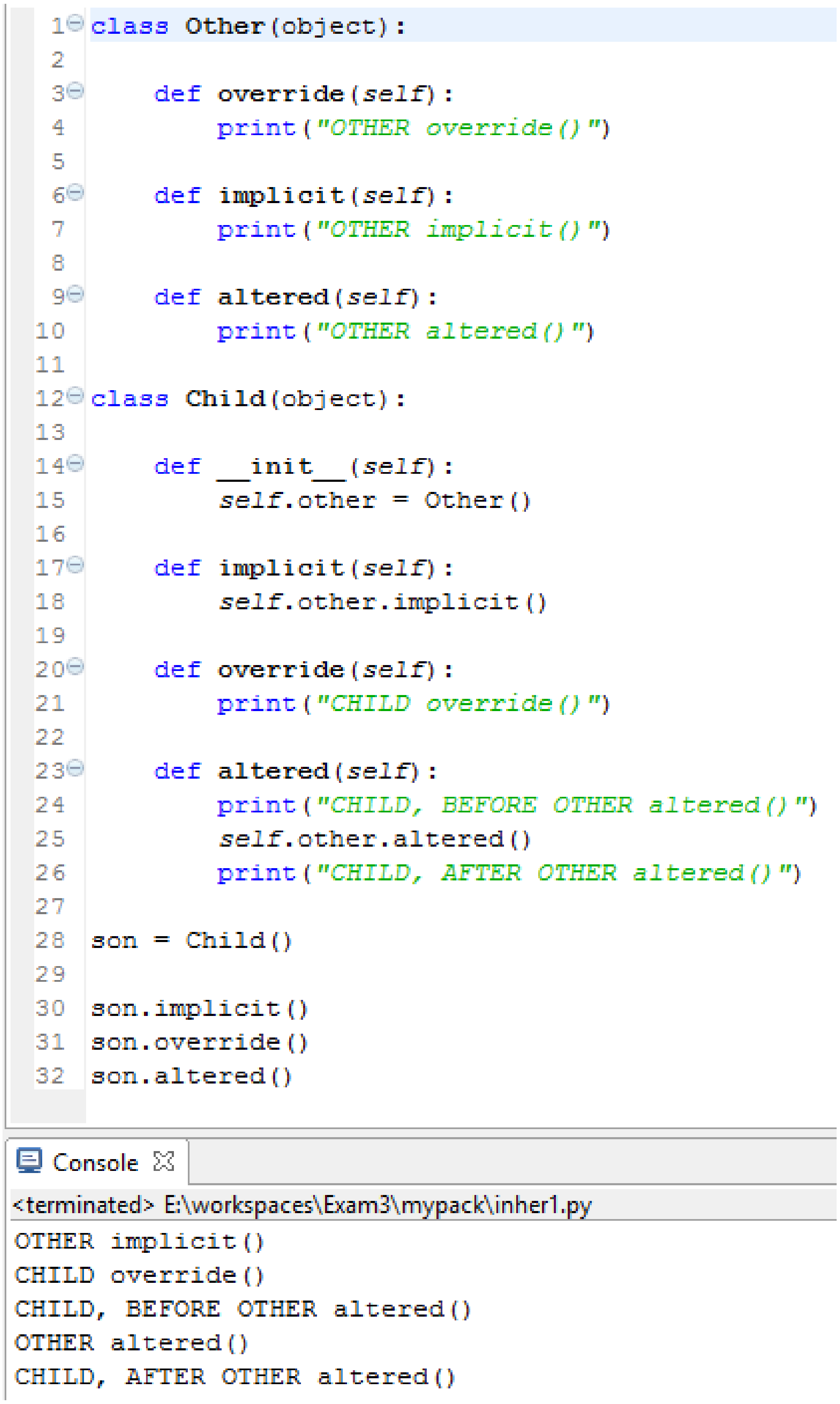


Console

```
<terminated> E:\workspaces\eXAM4\mypack\exam.py
Beef/Turkey BallPark is type <class '__main__.HotDog'>
  has 230 calories
  and tastes like Extremely processed meat
Lowfat Beef Patty is type <class '__main__.Hamburger'>
  has 260 calories
  and tastes like grilled goodness
Micky Mouse shaped Chicken Tenders is type <class '__main__.ChickenPatty'>
  has 170 calories
  and tastes like tastes like chicken
```

5. COMPOSITION

Inheritance is useful, but another way to do the exact same thing is just to *use* other classes and modules, rather than rely on implicit inheritance. If you look at the three ways to exploit inheritance, two of the three involve writing new code to replace or alter functionality. This can easily be replicated by just calling functions in a module. Here's an example of doing this:



```python
class Other(object):

    def override(self):
        print("OTHER override()")

    def implicit(self):
        print("OTHER implicit()")

    def altered(self):
        print("OTHER altered()")

class Child(object):

    def __init__(self):
        self.other = Other()

    def implicit(self):
        self.other.implicit()

    def override(self):
        print("CHILD override()")

    def altered(self):
        print("CHILD, BEFORE OTHER altered()")
        self.other.altered()
        print("CHILD, AFTER OTHER altered()")

son = Child()

son.implicit()
son.override()
son.altered()
```

Console

```
<terminated> E:\workspaces\Exam3\mypack\inher1.py
OTHER implicit()
CHILD override()
CHILD, BEFORE OTHER altered()
OTHER altered()
CHILD, AFTER OTHER altered()
```