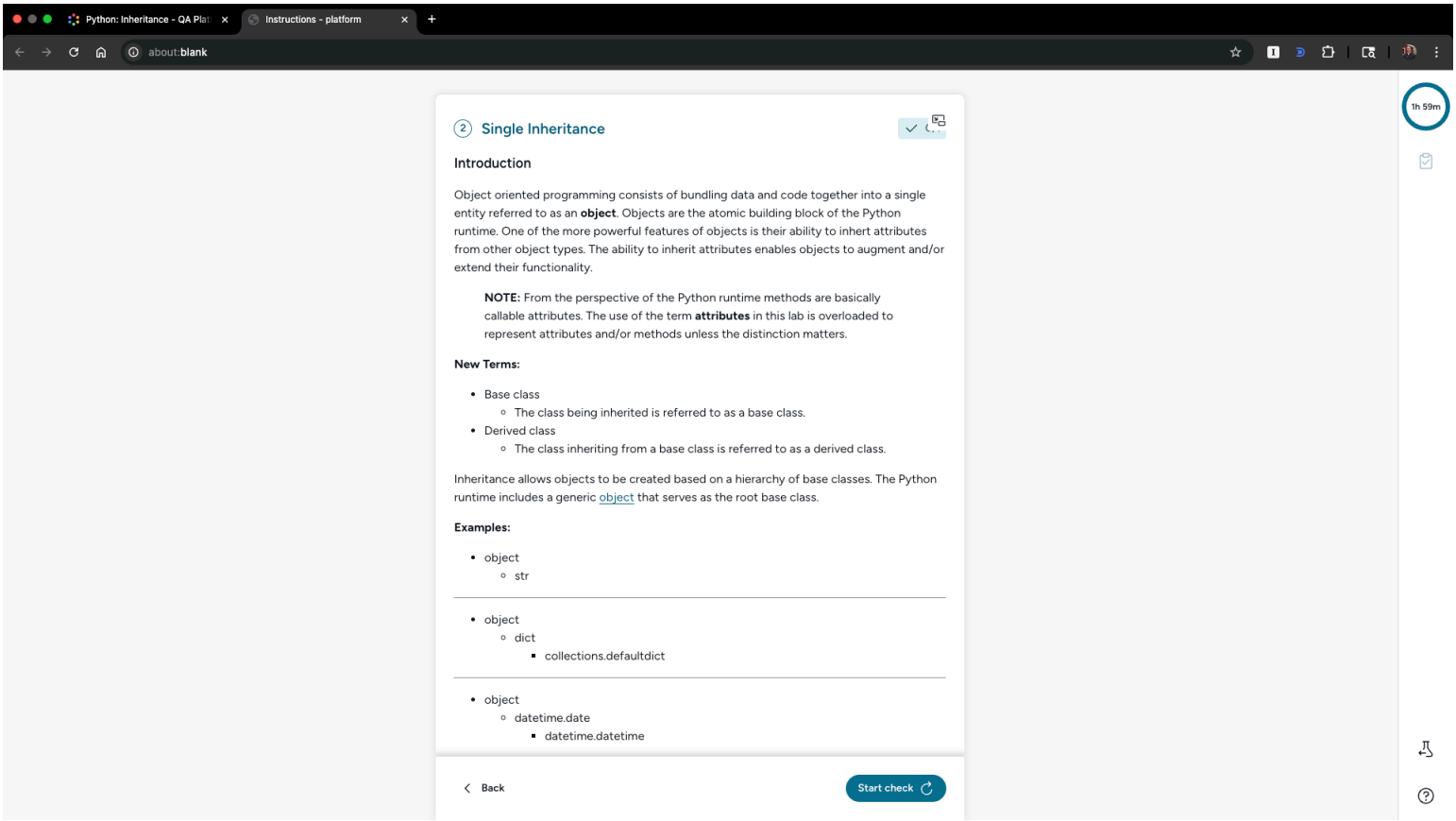
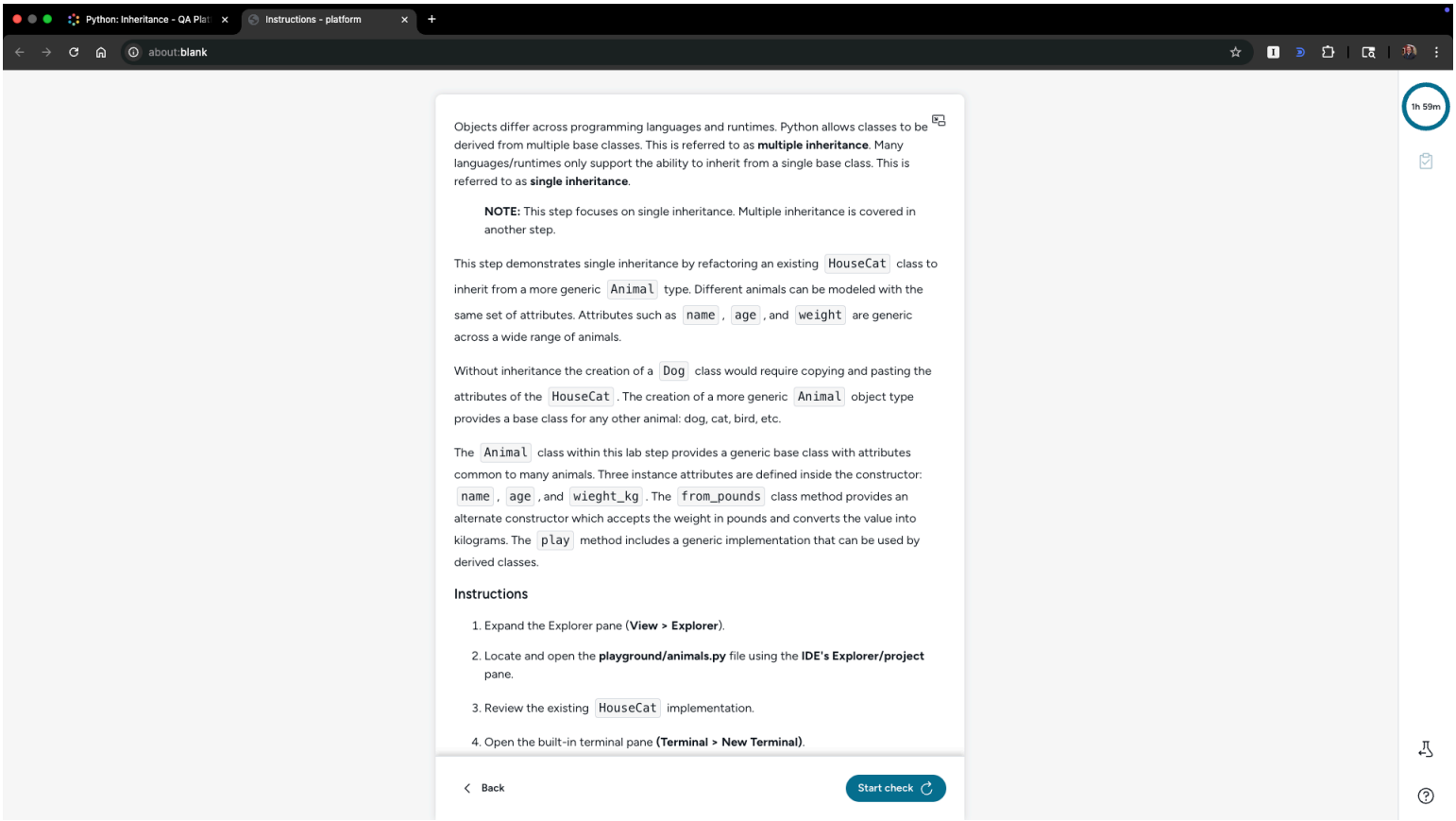
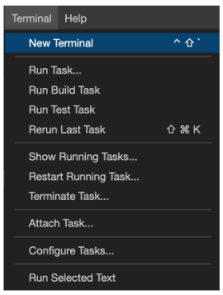

Python: Inheritance - QA Plat
Instructions - platform
about:blank
1h 59m
2 Single Inheritance
Introduction
Object oriented programming consists of bundling data and code together into a single entity referred to as an **object**. Objects are the atomic building block of the Python runtime. One of the more powerful features of objects is their ability to inherit attributes from other object types. The ability to inherit attributes enables objects to augment and/or extend their functionality.
NOTE: From the perspective of the Python runtime methods are basically callable attributes. The use of the term **attributes** in this lab is overloaded to represent attributes and/or methods unless the distinction matters.
New Terms:
Base class
The class being inherited is referred to as a base class.
Derived class
The class inheriting from a base class is referred to as a derived class.
Inheritance allows objects to be created based on a hierarchy of base classes. The Python runtime includes a generic object that serves as the root base class.
Examples:
object
str
object
dict
collections.defaultdict
object
datetime.date
datetime.datetime
Back
Start check


Python: Inheritance - QA Plat
Instructions - platform
about:blank
1h 59m
Objects differ across programming languages and runtimes. Python allows classes to be derived from multiple base classes. This is referred to as multiple inheritance. Many languages/runtimes only support the ability to inherit from a single base class. This is referred to as single inheritance.
NOTE: This step focuses on single inheritance. Multiple inheritance is covered in another step.
This step demonstrates single inheritance by refactoring an existing HouseCat class to inherit from a more generic Animal type. Different animals can be modeled with the same set of attributes. Attributes such as name, age, and weight are generic across a wide range of animals.
Without inheritance the creation of a Dog class would require copying and pasting the attributes of the HouseCat. The creation of a more generic Animal object type provides a base class for any other animal: dog, cat, bird, etc.
The Animal class within this lab step provides a generic base class with attributes common to many animals. Three instance attributes are defined inside the constructor: name, age, and wieght_kg. The from_pounds class method provides an alternate constructor which accepts the weight in pounds and converts the value into kilograms. The play method includes a generic implementation that can be used by derived classes.
Instructions
1. Expand the Explorer pane (View > Explorer).
2. Locate and open the playground/animals.py file using the IDE's Explorer/project pane.
3. Review the existing HouseCat implementation.
4. Open the built-in terminal pane (Terminal > New Terminal).
Back
Start check

Python: Inheritance - QA Plat | Instructions - platform | about:blank

4. Open the built-in terminal pane **(Terminal > New Terminal)**.



5. Run the code in the terminal prior to any changes to observe the output.

```
python3 playground/animals.py
```

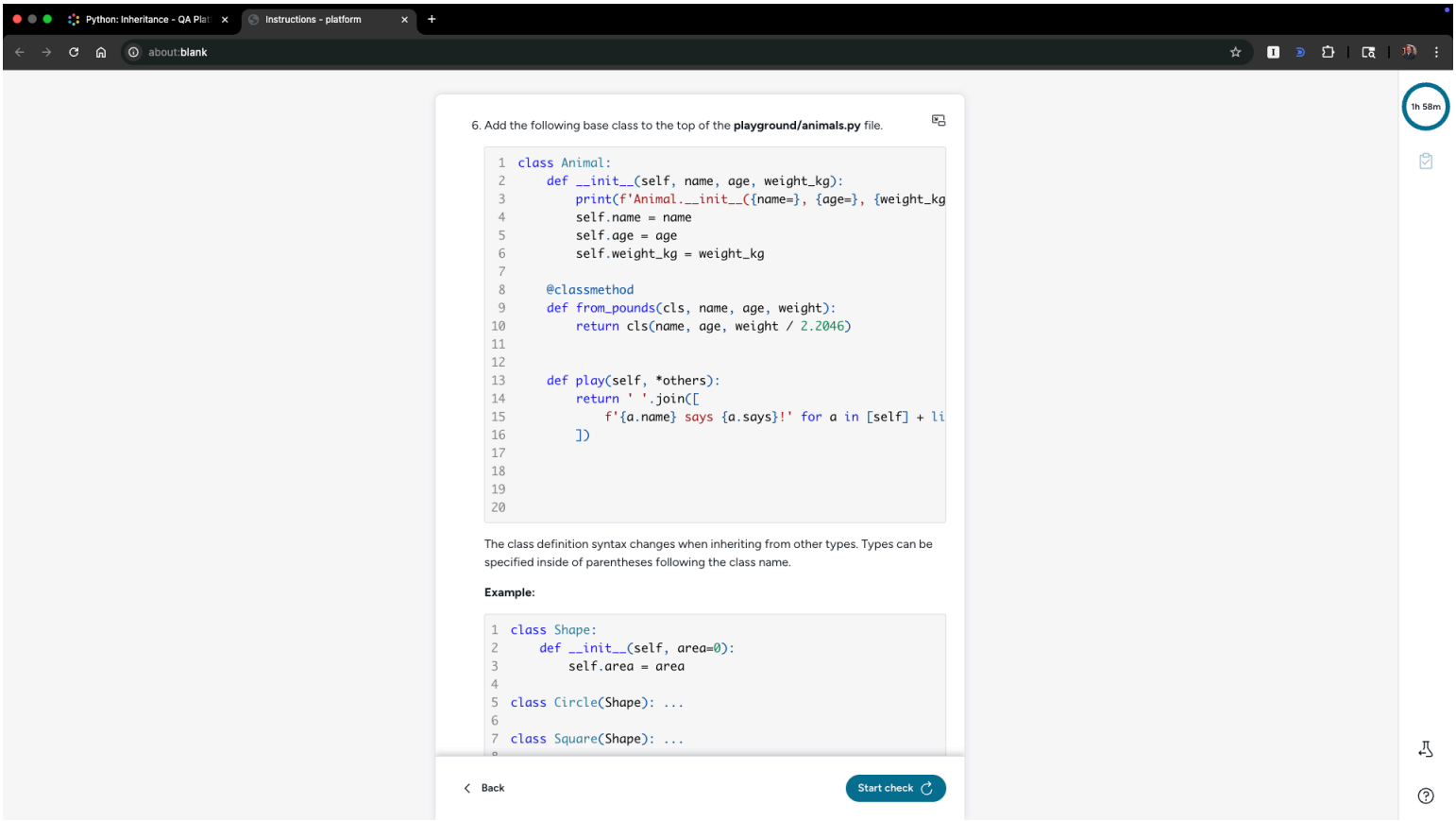
```
Meet Ada the 3 year old house cat.
Meet Gus the 4 year old house cat.
Meet Hal the 2 year old, 5.9 kg. house cat.
```

6. Add the following base class to the top of the **playground/animals.py** file.

```
class Animal:
    def __init__(self, name, age, weight_kg):
        print(f'Animal.__init__({name=}, {age=}, {weight_kg
        self.name = name
        self.age = age
        self.weight_kg = weight_kg

    @classmethod
```

Back | Start check

1h 58m


Python: Inheritance - QA Plat
Instructions - platform
about:blank
6. Add the following base class to the top of the playground/animals.py file.
class Animal:
def __init__(self, name, age, weight_kg):
print(f'Animal.__init__({name=}, {age=}, {weight_kg
self.name = name
self.age = age
self.weight_kg = weight_kg
@classmethod
def from_pounds(cls, name, age, weight):
return cls(name, age, weight / 2.2046)
def play(self, *others):
return ' '.join([
f'{a.name} says {a.says}!' for a in [self] + li
])
The class definition syntax changes when inheriting from other types. Types can be specified inside of parentheses following the class name.
Example:
class Shape:
def __init__(self, area=0):
self.area = area
class Circle(Shape): ...
class Square(Shape): ...
Back
Start check
1h 58m

Python: Inheritance - QA Plat | Instructions - platform

about:blank

1h 58m

```
5 class Circle(Shape): ...
6
7 class Square(Shape): ...
8
```

The above example demonstrates the `Circle` and `Square` classes inheriting from the `Shape` class. All attributes defined for the `Shape` class will exist for the `Circle` and `Square` classes.

7. Refactor the `HouseCat` to inherit from the `Animal` type.

```
1 class HouseCat(Animal):
2
```

```
class HouseCat(Animal):
    ''' Models a standard house cat. '''
```

The `Animal` class is now the **base class** for the `HouseCat` . Since the `HouseCat` derives attributes from its base class, it's no longer required to explicitly specify the same attributes.

8. Remove the `__init__` , `from_pounds` , and `play` methods from the `HouseCat` class.

The code has changed however, the functionality for this application remains mostly unchanged. One difference between this and the previous implementation is the addition of a `print` statement at the start of the `Animal.__init__` method. This `print` statement will be used to help identify which `__init__` method is called. Excluding this added context, the output remains unchanged.

```
class Animal:
    def __init__(self, name, age, weight_kg):
        print(f'Animal.__init__({name=}, {age=}, {weight_kg=})')
        self.name = name
```

Back

Start check

```python
class Animal:
    def __init__(self, name, age, weight_kg):
        print(f'Animal.__init__({name=}, {age=}, {weight_kg=})')
        self.name = name
        self.age = age
        self.weight_kg = weight_kg

    @classmethod
    def from_pounds(cls, name, age, weight):
        return cls(name, age, weight / 2.2046)


    def play(self, *others):
        return ' '.join([
            f'{a.name} says {a.says}!' for a in [self] + list(others)
        ])

class HouseCat(Animal):
    ''' Models a standard house cat. '''
    says = 'meow'


if __name__ == '__main__':
    ada = HouseCat('Ada', 3, 6.3)
    print(f'Meet {ada.name} the {ada.age} year old house cat.')

    gus = HouseCat('Gus', 4, 5.1)
    print(f'Meet {gus.name} the {gus.age} year old house cat. ')

    hal = HouseCat.from_pounds('Hal', 2, 13)
    print(f'Meet {hal.name} the {hal.age} year old, {hal.weight_kg:.1f} kg. house cat. ')
```

9. Run the code to observe the output.

```
python3 playground/animals.py
```

```
Animal.__init__(name='Ada', age=3, weight_kg=6.3)
Meet Ada the 3 year old house cat.
Animal.__init__(name='Gus', age=4, weight_kg=5.1)
Meet Gus the 4 year old house cat.
Animal.__init__(name='Hal', age=2, weight_kg=5.896761317245758)
Meet Hal the 2 year old, 5.9 kg. house cat.
```

The value of derived classes is the ability to augment and/or extend the functionality of the base class. The runtime allows derived classes to override methods of the base class.

Python: Inheritance - QA Plat | Instructions - platform | about:blank

Currently the `HouseCat` inherits its `__init__` method from the `Animal` base class. Explicitly adding methods to a derived class that already exist for the base class **overrides** which method implementation is called.

The below `__init__` method will override the implementation from the base class once added to the `HouseCat` class.

10. Add the following `__init__` method to the `HouseCat` class.

```
    def __init__(self, name, age, weight_kg, extrovertion_s
        print(f'HouseCat.__init__({name=}, {age=}, {weight_
        self.name = name
        self.age = age
        self.weight_kg = weight_kg
        # Set the level of introversion / extroversion
        # 0 = Introvert
        # 5 = Extrovert
        self.extrovertion_scale = extrovertion_scale


```

```
class HouseCat(Animal):
    ''' Models a standard house cat. '''
    says = 'meow'

    def __init__(self, name, age, weight_kg, extrovertion_scale=3):
        print(f'HouseCat.__init__({name=}, {age=}, {weight_kg=}, {extrovertion_scale=})')
        self.name = name
        self.age = age
        self.weight_kg = weight_kg
        # Set the level of introversion / extroversion
        # 0 = Introvert
        # 5 = Extrovert
        self.extrovertion_scale = extrovertion_scale
```

11. Run the code to observe the constructor from the `HouseCat` being called.

```
python3 playground/animals.py
```

Back | Start check

1h 58m

Python: Inheritance - QA Plat | Instructions - platform | about:blank

11. Run the code to observe the constructor from the `HouseCat` being called.

```
python3 playground/animals.py
```

```
HouseCat.__init__(name='Ada', age=3, weight_kg=6.3, extrovertion_scale=3)
Meet Ada the 3 year old house cat.
HouseCat.__init__(name='Gus', age=4, weight_kg=5.1, extrovertion_scale=3)
Meet Gus the 4 year old house cat.
HouseCat.__init__(name='Hal', age=2, weight_kg=5.89676131724575 8, extrovertion_scale=3)
Meet Hal the 2 year old, 5.9 kg. house cat.
```

Method overrides can be used to change method signatures and/or update/extend methods. The above `__init__` method is extended and given a new method signature.

Derived classes require access to the attributes of base classes in order to more fully extend functionality. The standard library includes a callable used to provide access to base classes. The built-in `super` callable returns an object's base class. Base class access allows derived classes to interact with the attributes of base classes.

> **NOTE:** The `super` callable is a powerful mechanism for delegating method calls to base classes in the inheritance hierarchy. It will be covered more in-depth in another step.

The `HouseCat.__init__` method overrides the implementation from the `Animal` base class, preventing it from being called. The use of the `super` callable allows the constructor — or any other method — of the base class to be called. Notice the below code explicitly calls the `__init__` method of the `Animal` base class.

12. Replace the `__init__` method for the `HouseCat` class with the following implementation.

```
    def __init__(self, name, age, weight_kg, extrovertion_sc
        print(f'HouseCat.__init__({name=}, {age=}, {weight_k
        super().__init__(name, age, weight_kg)
```

Back | Start check

1h 58m

Python: Inheritance - QA Plat | Instructions - platform | about:blank

1h 58m

12. Replace the `__init__` method for the `HouseCat` class with the following implementation.

```
    def __init__(self, name, age, weight_kg, extrovertion_sc
        print(f'HouseCat.__init__({name=}, {age=}, {weight_k
        super().__init__(name, age, weight_kg)
        # Set the level of introversion / extroversion
        # 0 = Introvert
        # 5 = Extrovert
        self.extrovertion_scale = extrovertion_scale

```

13. Run the code to observe the revised output.

```
python3 playground/animals.py
```

```
HouseCat.__init__(name='Ada', age=3, weight_kg=6.3, extrovertion_scale=3)
Animal.__init__(name='Ada', age=3, weight_kg=6.3)
Meet Ada the 3 year old house cat.
HouseCat.__init__(name='Gus', age=4, weight_kg=5.1, extrovertion_scale=3)
Animal.__init__(name='Gus', age=4, weight_kg=5.1)
Meet Gus the 4 year old house cat.
HouseCat.__init__(name='Hal', age=2, weight_kg=5.896761317245758, extrovertion_scale=3)
Animal.__init__(name='Hal', age=2, weight_kg=5.896761317245758)
Meet Hal the 2 year old, 5.9 kg. house cat.
```

Re-using existing implementations via the `super` callable has benefits such as: reduced code duplication, easier code refactoring, etc. The below `HouseCat.play` method overrides the base class implementation. The base class implementation is conditionally called by the derived class.

14. Add the following `play` method to the `HouseCat` class.

```
    def play(self, *others):
        if self.extrovertion_scale <= 2:
            return f'{self.name} wants to be alone right now
        return super().play(
            *[animal for animal in others if getattr(animal,
```

Back | Start check

Python: Inheritance - QA Plat | Instructions - platform | about:blank

1h 58m

14. Add the following `play` method to the `HouseCat` class.

```
    def play(self, *others):
        if self.extrovertion_scale <= 2:
            return f'{self.name} wants to be alone right now
        return super().play(
            *[animal for animal in others if getattr(animal,
        )
```

```
class HouseCat(Animal):
    ''' Models a standard house cat. '''
    says = 'meow'

    def __init__(self, name, age, weight_kg, extrovertion_scale=3):
        print(f'HouseCat.__init__({name=}, {age=}, {weight_kg=}, {extrovertion_scale=})')
        super().__init__(name, age, weight_kg)
        # Set the level of introversion / extroversion
        # 0 = Introvert
        # 5 = Extrovert
        self.extrovertion_scale = extrovertion_scale


    def play(self, *others):
        if self.extrovertion_scale <= 2:
            return f'{self.name} wants to be alone right now.'
        return super().play(
            *[animal for animal in others if getattr(animal, 'extrovertion_scale', 3) > 2]
        )
```

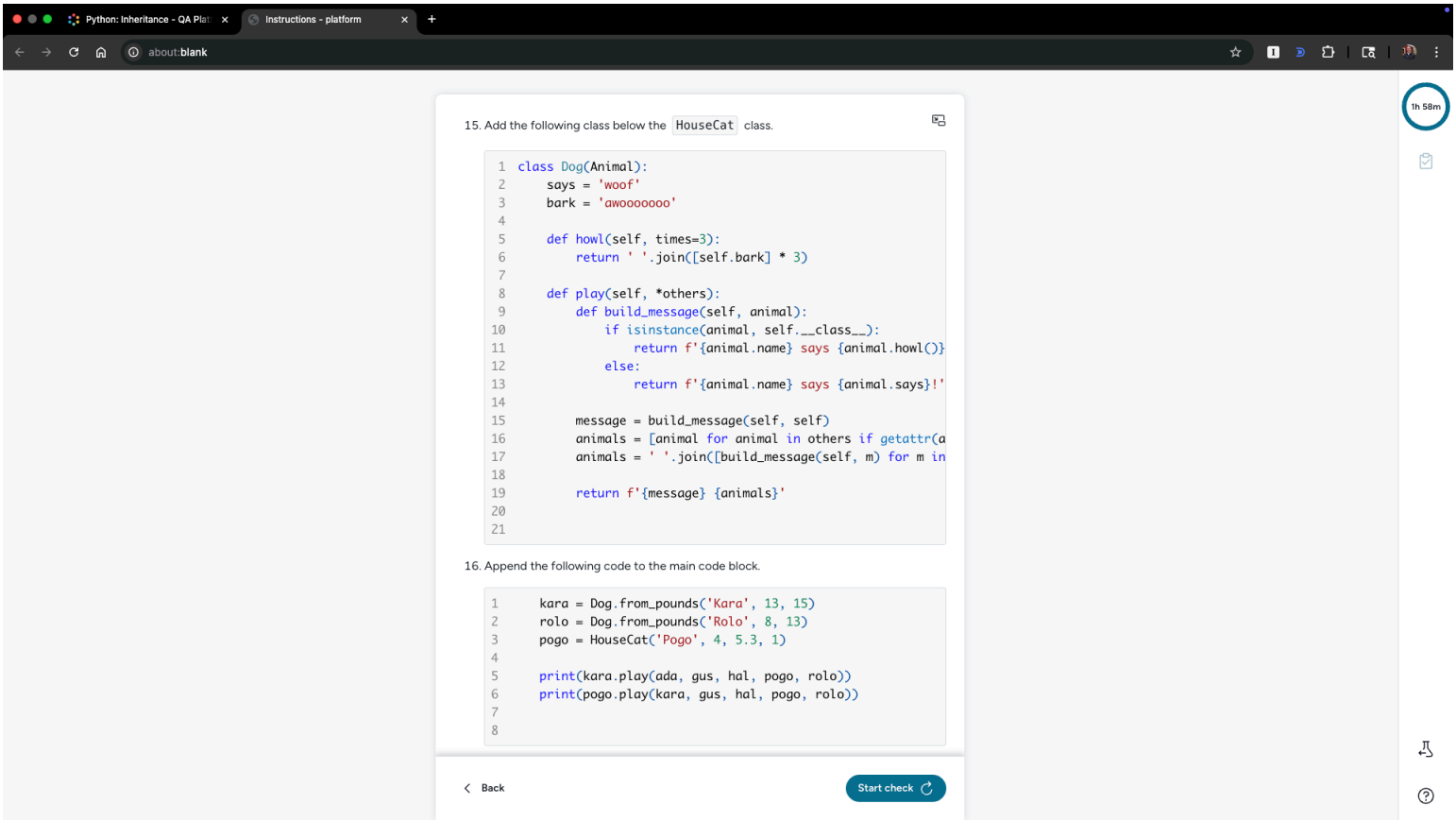
Base classes provide a means of sharing functionality common to multiple derived classes. The below code creates a derived `Dog` class.

15. Add the following class below the `HouseCat` class.

```
class Dog(Animal):
    says = 'woof'
    bark = 'awooooooo'

```

Back

Start check


Python: Inheritance - QA Plat
Instructions - platform
about:blank
15. Add the following class below the HouseCat class.
class Dog(Animal):
says = 'woof'
bark = 'awooooooo'
def howl(self, times=3):
return ' '.join([self.bark] * 3)
def play(self, *others):
def build_message(self, animal):
if isinstance(animal, self.__class__):
return f'{animal.name} says {animal.howl()}
else:
return f'{animal.name} says {animal.says}!'
message = build_message(self, self)
animals = [animal for animal in others if getattr(a
animals = ' '.join([build_message(self, m) for m in
return f'{message} {animals}'
16. Append the following code to the main code block.
kara = Dog.from_pounds('Kara', 13, 15)
rolo = Dog.from_pounds('Rolo', 8, 13)
pogo = HouseCat('Pogo', 4, 5.3, 1)
print(kara.play(ada, gus, hal, pogo, rolo))
print(pogo.play(kara, gus, hal, pogo, rolo))
Back
Start check
1h 58m

```python
if __name__ == '__main__':
    ada = HouseCat('Ada', 3, 6.3)
    print(f'Meet {ada.name} the {ada.age} year old house cat.')

    gus = HouseCat('Gus', 4, 5.1)
    print(f'Meet {gus.name} the {gus.age} year old house cat. ')

    hal = HouseCat.from_pounds('Hal', 2, 13)
    print(f'Meet {hal.name} the {hal.age} year old, {hal.weight_kg:.1f} kg. house cat. ')

    kara = Dog.from_pounds('Kara', 13, 15)
    rolo = Dog.from_pounds('Rolo', 8, 13)
    pogo = HouseCat('Pogo', 4, 5.3, 1)

    print(kara.play(ada, gus, hal, pogo, rolo))
    print(pogo.play(kara, gus, hal, pogo, rolo))
```

17. Run the code to observe the newly created `Dog` class.

```
Animal.__init__(name='Kara', age=13, weight_kg=6.803955390457748)
Animal.__init__(name='Rolo', age=8, weight_kg=5.896763117243758)
HouseCat.__init__(name='Pogo', age=4, weight_kg=5.3, extrovertion_scale=1)
Animal.__init__(name='Pogo', age=4, weight_kg=5.3)
Kara says awoooooooo awoooooooo awooooooo! Ada says meow! Gus says meow! Hal says meow! Rolo says awoooooooo awoooooooo awooooooo!
Pogo wants to be alone right now.
```

The relationship between base classes and derived classes is commonly referred to as an: **is a** relationship. For example: a `HouseCat` **is an** `Animal` and a `Dog` **is an** Animal.

Derived classes can be used in place of base classes since they inherit all the same attributes. This enables code to be written in a more generic manner.

18. Add the following code above the main code block and under the class definitions.

```python
def play_with(person: str, animals: list[Animal]):
    print(f'{person} is playing with: {" ".join([a.play() fo
```

Because the `play_with` function expects a list of `Animals` both the `Dog` and `HouseCat` types can be used. Although derived classes may contain their own unique attributes, the function only uses attributes available to the `Animal` base

Python: Inheritance - QA Plat | Instructions - platform | about:blank

1h 58m

Because the `play_with` function expects a list of `Animals` both the `Dog` and `HouseCat` types can be used. Although derived classes may contain their own unique attributes, the function only uses attributes available to the `Animal` base class. This enables callables to operate on many different types of objects with shared base classes.

19. Append the following code to the main code block.

```
    play_with('This developer', [kara, gus, hal, pogo, rolo,


```

20. Run the code to observe the results.

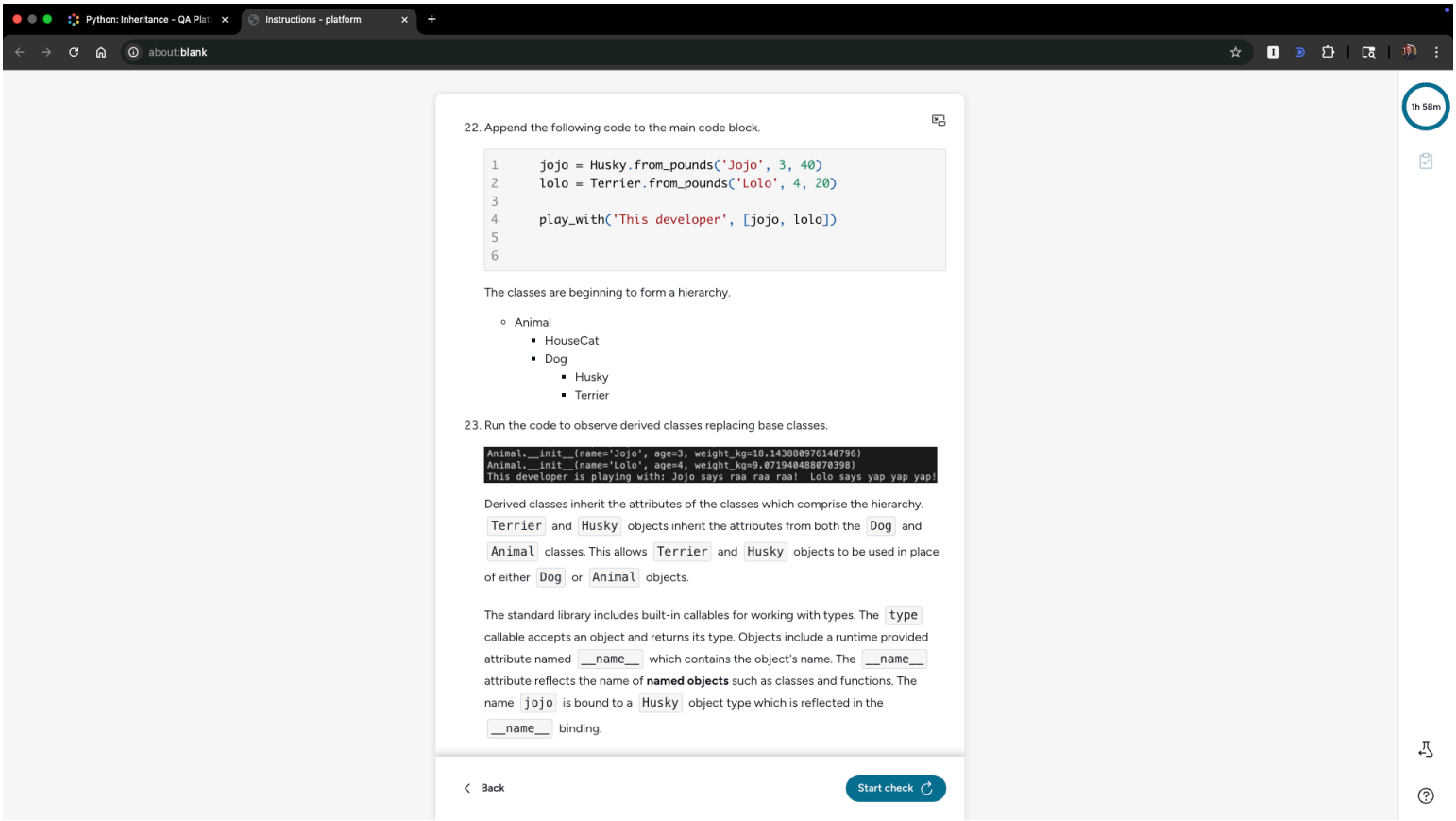
Complex type hierarchies can be created by using derived classes as base classes.

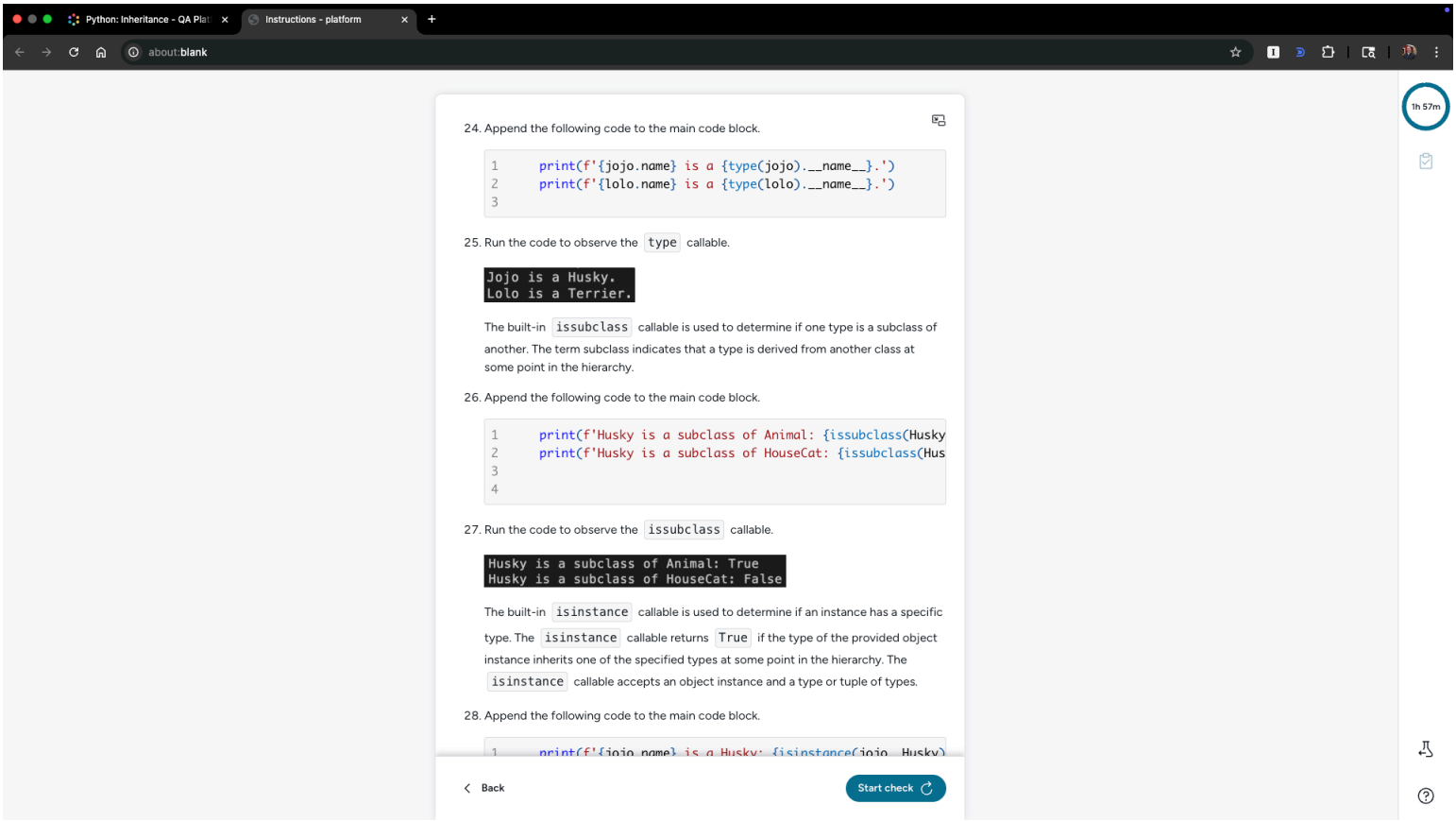
21. Add the following class below the `Dog` class.

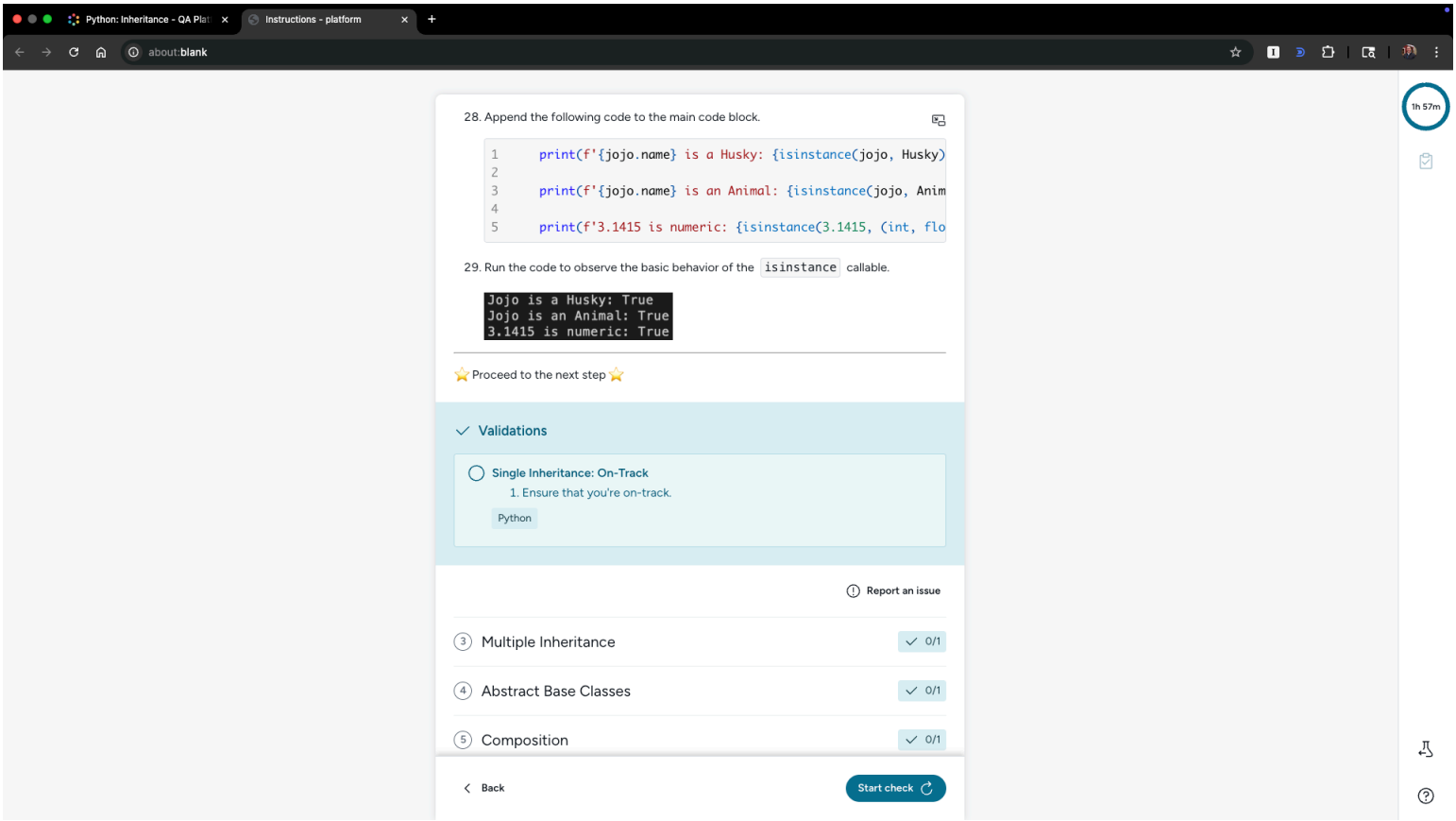
```
class Husky(Dog):
    says = 'woo woo'
    bark = 'raa'

class Terrier(Dog):
    says = 'yip'
    bark = 'yap'


```

22. Append the following code to the main code block.

Back | Start check


Python: Inheritance - QA Plat
Instructions - platform
about:blank
1h 58m
22. Append the following code to the main code block.
jojo = Husky.from_pounds('Jojo', 3, 40)
lolo = Terrier.from_pounds('Lolo', 4, 20)
play_with('This developer', [jojo, lolo])
The classes are beginning to form a hierarchy.
Animal
HouseCat
Dog
Husky
Terrier
23. Run the code to observe derived classes replacing base classes.
Animal.__init__(name='Jojo', age=3, weight_kg=18.143880976140796)
Animal.__init__(name='Lolo', age=4, weight_kg=9.071940488070398)
This developer is playing with: Jojo says raa raa raa! Lolo says yap yap yap!
Derived classes inherit the attributes of the classes which comprise the hierarchy. Terrier and Husky objects inherit the attributes from both the Dog and Animal classes. This allows Terrier and Husky objects to be used in place of either Dog or Animal objects.
The standard library includes built-in callables for working with types. The type callable accepts an object and returns its type. Objects include a runtime provided attribute named __name__ which contains the object's name. The __name__ attribute reflects the name of named objects such as classes and functions. The name jojo is bound to a Husky object type which is reflected in the __name__ binding.
Back
Start check


Python: Inheritance - QA Plat
Instructions - platform
about:blank
1h 57m
24. Append the following code to the main code block.
1 print(f'{jojo.name} is a {type(jojo).__name__}.')
2 print(f'{lolo.name} is a {type(lolo).__name__}.')
3
25. Run the code to observe the type callable.
Jojo is a Husky.
Lolo is a Terrier.
The built-in issubclass callable is used to determine if one type is a subclass of another. The term subclass indicates that a type is derived from another class at some point in the hierarchy.
26. Append the following code to the main code block.
1 print(f'Husky is a subclass of Animal: {issubclass(Husky
2 print(f'Husky is a subclass of HouseCat: {issubclass(Hus
3
4
27. Run the code to observe the issubclass callable.
Husky is a subclass of Animal: True
Husky is a subclass of HouseCat: False
The built-in isinstance callable is used to determine if an instance has a specific type. The isinstance callable returns True if the type of the provided object instance inherits one of the specified types at some point in the hierarchy. The isinstance callable accepts an object instance and a type or tuple of types.
28. Append the following code to the main code block.
Back
Start check


Python: Inheritance - QA Plat
Instructions - platform
about:blank
28. Append the following code to the main code block.
1 print(f'{jojo.name} is a Husky: {isinstance(jojo, Husky)
2
3 print(f'{jojo.name} is an Animal: {isinstance(jojo, Anim
4
5 print(f'3.1415 is numeric: {isinstance(3.1415, (int, flo
29. Run the code to observe the basic behavior of the isinstance callable.
Jojo is a Husky: True
Jojo is an Animal: True
3.1415 is numeric: True
Proceed to the next step
Validations
Single Inheritance: On-Track
1. Ensure that you're on-track.
Python
Report an issue
3 Multiple Inheritance 0/1
4 Abstract Base Classes 0/1
5 Composition 0/1
Back
Start check
1h 57m