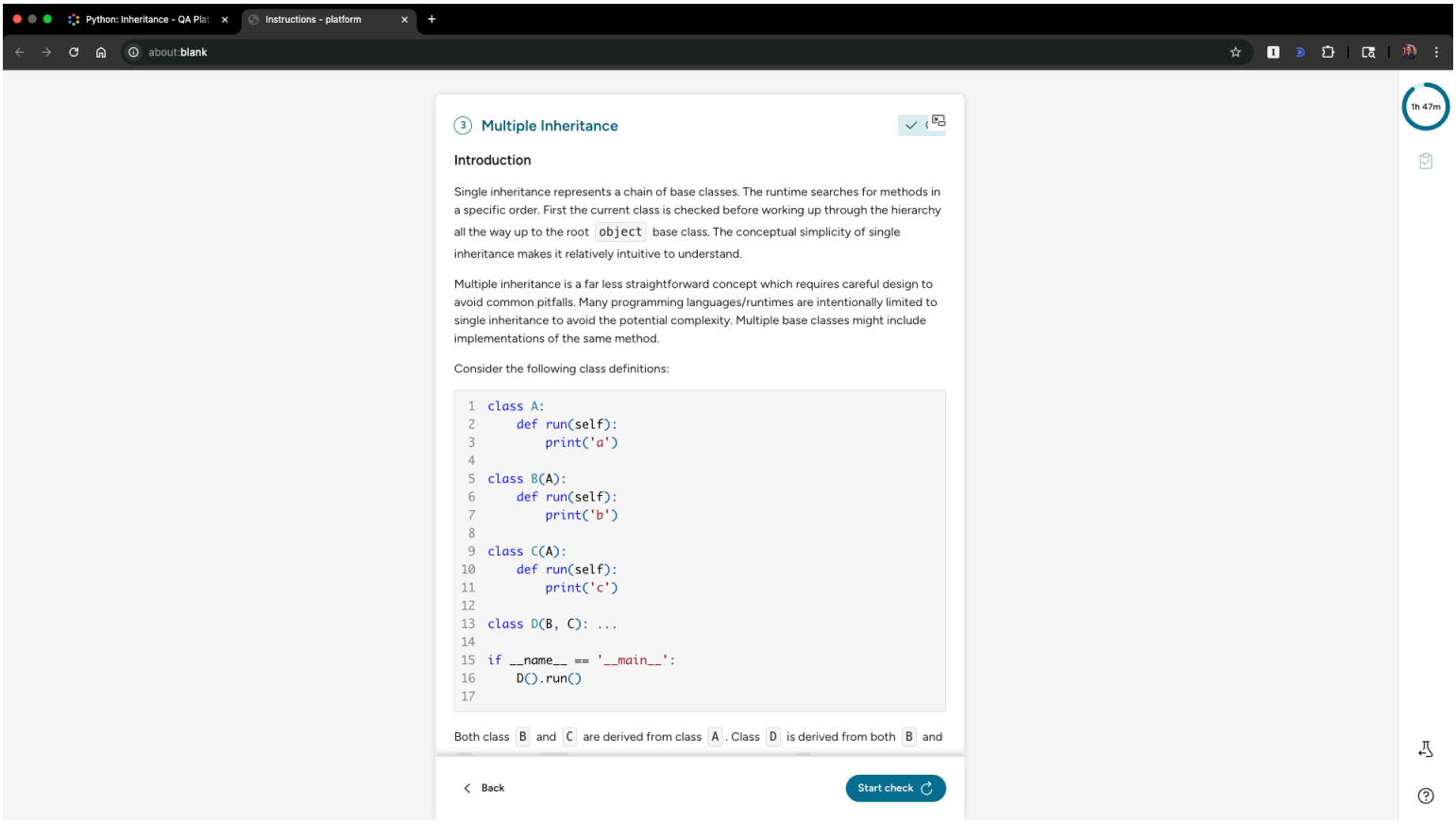

Python: Inheritance - QA Plat
Instructions - platform
about:blank
1h 47m
3 Multiple Inheritance
Introduction
Single inheritance represents a chain of base classes. The runtime searches for methods in a specific order. First the current class is checked before working up through the hierarchy all the way up to the root object base class. The conceptual simplicity of single inheritance makes it relatively intuitive to understand.
Multiple inheritance is a far less straightforward concept which requires careful design to avoid common pitfalls. Many programming languages/runtimes are intentionally limited to single inheritance to avoid the potential complexity. Multiple base classes might include implementations of the same method.
Consider the following class definitions:
1 class A:
2 def run(self):
3 print('a')
4
5 class B(A):
6 def run(self):
7 print('b')
8
9 class C(A):
10 def run(self):
11 print('c')
12
13 class D(B, C): ...
14
15 if __name__ == '__main__':
16 D().run()
17
Both class B and C are derived from class A . Class D is derived from both B and
Back
Start check

Python: Inheritance - QA Plat | Instructions - platform

about:blank

1h 47m

Both class `B` and `C` are derived from class `A`. Class `D` is derived from both `B` and `C`. When the `run` method is called for an instance of class `D` which implementation of `run` should be called? This is commonly referred to as the diamond problem.

Instructions

1. Open and review the **playground/multi.py** file.
2. Run the code to observe which `run` implementation is called.

```
python3 playground/multi.py
```

```
b
```

Notice the runtime called the `run` method from the `B` class. The runtime includes logic to determine the order that base classes are checked for a requested method. This is referred to as the **method resolution order**, commonly abbreviated **mro**. Every object type includes a list of base classes representing the method resolution order.

The runtime includes mechanisms used to determine the resolution order. Object types include a class method named `mro` and a class attribute named `__mro__`. The `mro` method returns a list of base classes in resolution order. The `__mro__` attribute includes the same information as a tuple.

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

```
    print('as method', D.mro())
    print('attribute', D.__mro__)

if __name__ == '__main__':
```

Back

Start check

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

```
if __name__ == '__main__':
    D().run()
    print('as method', D.mro())
    print('attribute', D.__mro__)
```

4. Run the code to observe the method resolution order.

```
as method [<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
attribute (<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>)
```

Notice the order of the method resolution. The runtime first checks for methods on the calling class before moving through the other classes. The runtime uses the C3 linearization algorithm to determine the method resolution order.

Guido van Rossum the creator of Python describes C3 thusly:

> Basically, the idea behind C3 is that if you write down all of the ordering rules imposed by inheritance relationships in a complex class hierarchy, the algorithm will determine a monotonic ordering of the classes that satisfies all of them. If such an ordering can not be determined, the algorithm will fail.

While the C3 algorithm is outside the scope of this lab, here's a summary of how it impacts the method resolution order.

- Derived classes are checked before base classes.
- Classes inheriting from multiple base classes maintain the base class order specified in the class definition.
- Classes are never repeated.

5. Replace the class definition for `D` with the following code.

```
1 class D(C, B): ...
2
```

```
class A:
    def run(self):
        print('a')
```

Back | Start check

1h 47m

```
class A:
    def run(self):
        print('a')

class B(A):
    def run(self):
        print('b')

class C(A):
    def run(self):
        print('c')

class D(C, B): ...
```

6. Run the code to observe the revised method resolution order.

```
as method [<class '__main__.D'>, <class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
attribute [<class '__main__.D'>, <class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
```

Notice that classes `C` and `B` have changed places to reflect the revised class definition order. Also notice that the `A` class is only specified once even though both `B` and `C` inherit from the `A` class.

Multiple inheritance is omitted in many other programming languages/runtimes due to the potential method resolution ambiguity. Effective multiple inheritance requires careful design in order to avoid tangled object hierarchies that are difficult to maintain. Two common use cases for multiple inheritance include **method delegation** and **mixins**.

Method delegation can assist with the diamond problem by delegating method calls to base classes. The built-in `super` callable used with single inheritance returns the base class. When used with multiple inheritance the `super` callable demonstrates its true power.

Consider the following goal: create a dictionary that normalizes keys as lowercase

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

1h 47m

Consider the following goal: create a dictionary that normalizes keys as lowercase alphanumeric characters. This can be easily accomplished by using the built-in `dict` type as a base class.

The below `NormalizedDict` class overrides the `__setitem__` and `__getitem__` magic methods and delegates to the base class.

> New to magic methods? Learn how to make objects behave more like built-in objects using magic methods.

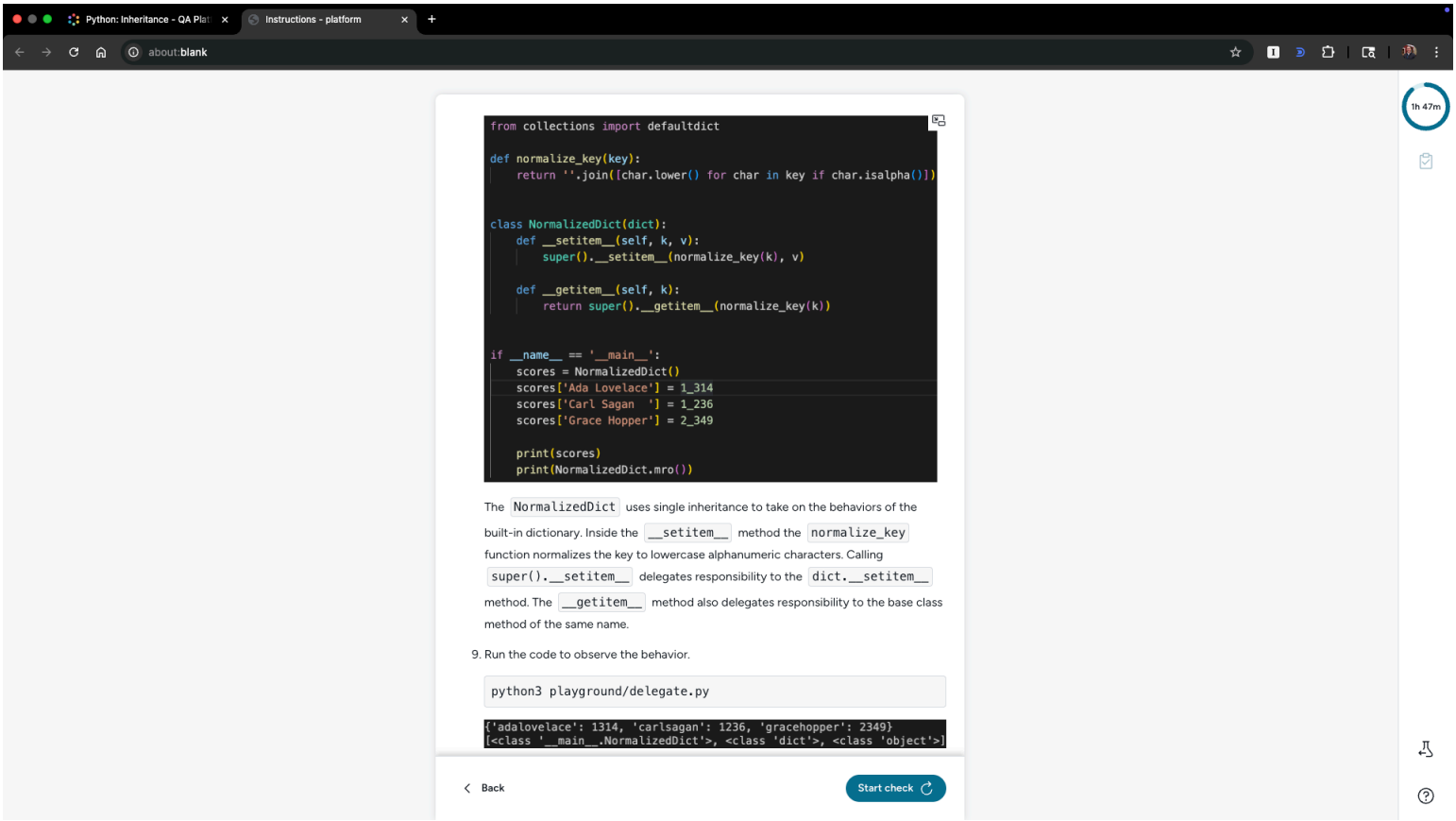
7. Open and review the **playground/delegate.py** file.
8. Append the following code above the main code block.

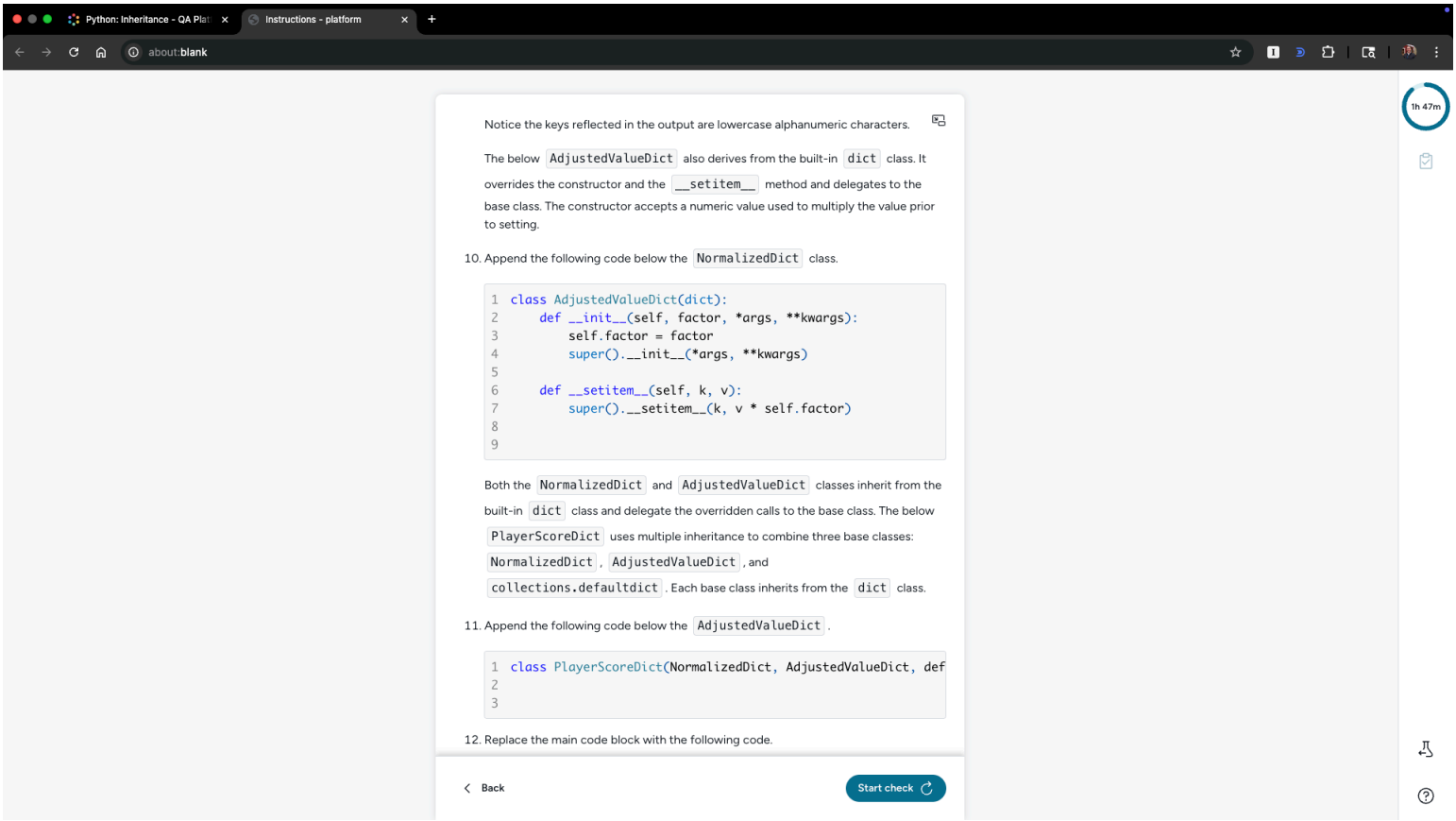
```
class NormalizedDict(dict):
    def __setitem__(self, k, v):
        super().__setitem__(normalize_key(k), v)

    def __getitem__(self, k):
        return super().__getitem__(normalize_key(k))


```

```
from collections import defaultdict

def normalize_key(key):
    return ''.join([char.lower() for char in key if char.isalpha()])


class NormalizedDict(dict):
    def __setitem__(self, k, v):
        super().__setitem__(normalize_key(k), v)

    def __getitem__(self, k):
        return super().__getitem__(normalize_key(k))
```

Back

Start check


Python: Inheritance - QA Plat
Instructions - platform
about:blank
1h 47m
from collections import defaultdict
def normalize_key(key):
return ''.join([char.lower() for char in key if char.isalpha()])
class NormalizedDict(dict):
def __setitem__(self, k, v):
super().__setitem__(normalize_key(k), v)
def __getitem__(self, k):
return super().__getitem__(normalize_key(k))
if __name__ == '__main__':
scores = NormalizedDict()
scores['Ada Lovelace'] = 1_314
scores['Carl Sagan '] = 1_236
scores['Grace Hopper'] = 2_349
print(scores)
print(NormalizedDict.mro())
The NormalizedDict uses single inheritance to take on the behaviors of the built-in dictionary. Inside the __setitem__ method the normalize_key function normalizes the key to lowercase alphanumeric characters. Calling super().__setitem__ delegates responsibility to the dict.__setitem__ method. The __getitem__ method also delegates responsibility to the base class method of the same name.
9. Run the code to observe the behavior.
python3 playground/delegate.py
{'adalovelace': 1314, 'carlsagan': 1236, 'gracehopper': 2349}
[<class '__main__.NormalizedDict'>, <class 'dict'>, <class 'object'>]
Back
Start check



Notice the keys reflected in the output are lowercase alphanumeric characters.

The below `AdjustedValueDict` also derives from the built-in `dict` class. It overrides the constructor and the `__setitem__` method and delegates to the base class. The constructor accepts a numeric value used to multiply the value prior to setting.

10. Append the following code below the `NormalizedDict` class.

```
class AdjustedValueDict(dict):
    def __init__(self, factor, *args, **kwargs):
        self.factor = factor
        super().__init__(*args, **kwargs)

    def __setitem__(self, k, v):
        super().__setitem__(k, v * self.factor)


```

Both the `NormalizedDict` and `AdjustedValueDict` classes inherit from the built-in `dict` class and delegate the overridden calls to the base class. The below `PlayerScoreDict` uses multiple inheritance to combine three base classes: `NormalizedDict`, `AdjustedValueDict`, and `collections.defaultdict`. Each base class inherits from the `dict` class.

11. Append the following code below the `AdjustedValueDict`.

```
class PlayerScoreDict(NormalizedDict, AdjustedValueDict, def


```

12. Replace the main code block with the following code.

Python: Inheritance - QA Plat | Instructions - platform

about:blank

12. Replace the main code block with the following code.

```
if __name__ == '__main__':
    scores = PlayerScoreDict(1_000, int)
    scores['Ada Lovelace'] = 1_314
    scores['Carl Sagan  '] = 1_236
    scores['Grace Hopper'] = 2_349

    print(scores)
    print('The value for MISSING is: ', scores['MISSING'])


```

```
from collections import defaultdict

def normalize_key(key):
    return ''.join([char.lower() for char in key if char.isalpha()])


class NormalizedDict(dict):
    def __setitem__(self, k, v):
        super().__setitem__(normalize_key(k), v)

    def __getitem__(self, k):
        return super().__getitem__(normalize_key(k))

class AdjustedValueDict(dict):
    def __init__(self, factor, *args, **kwargs):
        self.factor = factor
        super().__init__(*args, **kwargs)

    def __setitem__(self, k, v):
        super().__setitem__(k, v * self.factor)

class PlayerScoreDict(NormalizedDict, AdjustedValueDict, defaultdict): ...


if __name__ == '__main__':
    scores = PlayerScoreDict(1_000, int)
    scores['Ada Lovelace'] = 1_314
```

Back

Start check

1h 47m

```python
from collections import defaultdict

def normalize_key(key):
    return ''.join([char.lower() for char in key if char.isalpha()])


class NormalizedDict(dict):
    def __setitem__(self, k, v):
        super().__setitem__(normalize_key(k), v)

    def __getitem__(self, k):
        return super().__getitem__(normalize_key(k))

class AdjustedValueDict(dict):
    def __init__(self, factor, *args, **kwargs):
        self.factor = factor
        super().__init__(*args, **kwargs)

    def __setitem__(self, k, v):
        super().__setitem__(k, v * self.factor)

class PlayerScoreDict(NormalizedDict, AdjustedValueDict, defaultdict): ...


if __name__ == '__main__':
    scores = PlayerScoreDict(1_000, int)
    scores['Ada Lovelace'] = 1_314
    scores['Carl Sagan  '] = 1_236
    scores['Grace Hopper'] = 2_349

    print(scores)
    print('The value for MISSING is: ', scores['MISSING'])
```

The `PlayerScoreDict` inherits from three base classes which all have the same shared base class. When this code is run, what is the expected output?

13. Run the code to observe the results.

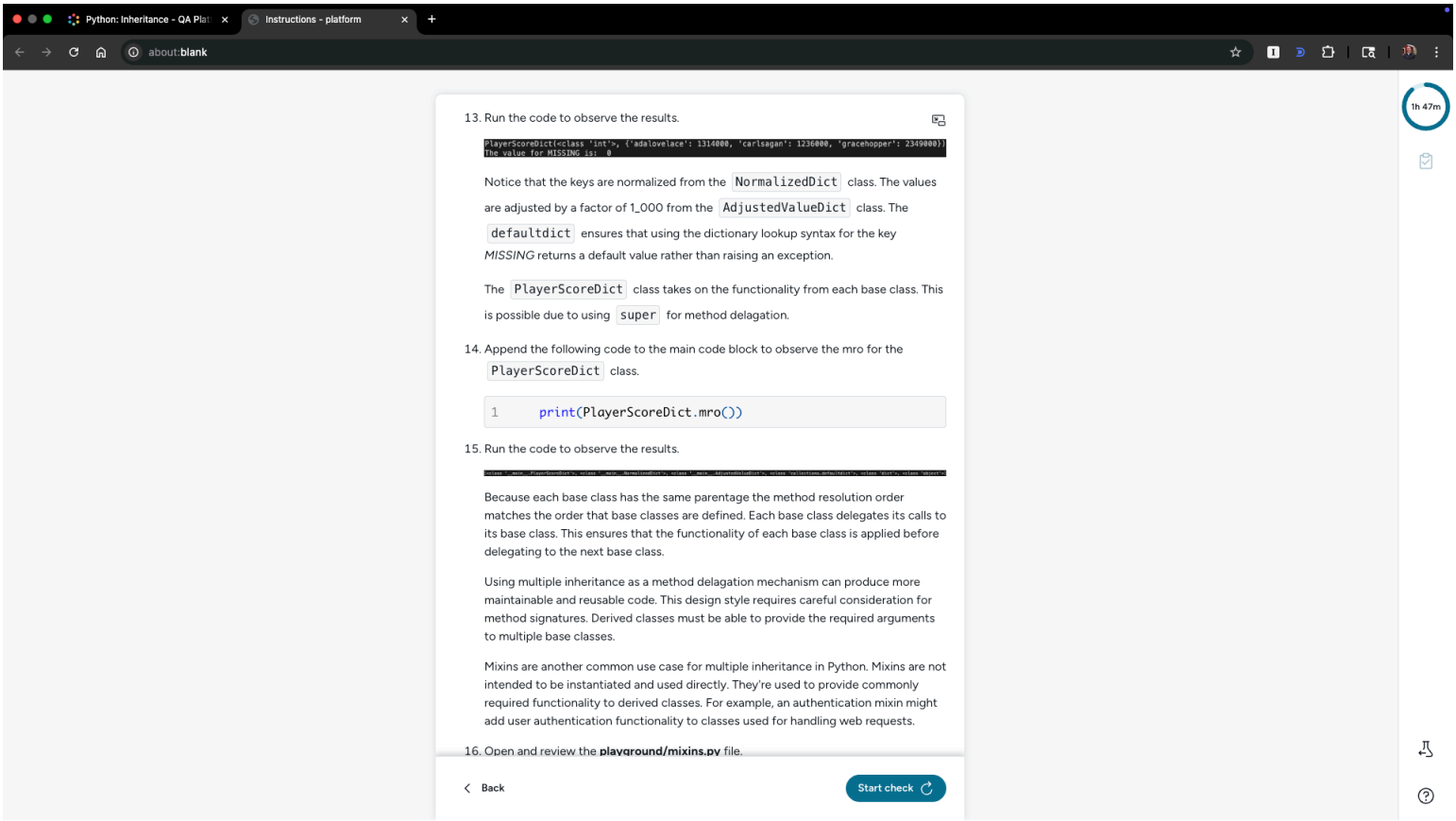
```
PlayerScoreDict(<class 'int'>, {'adalovelace': 1314000, 'carlsagan': 1236000, 'gracehopper': 2349000})
The value for MISSING is:  0
```

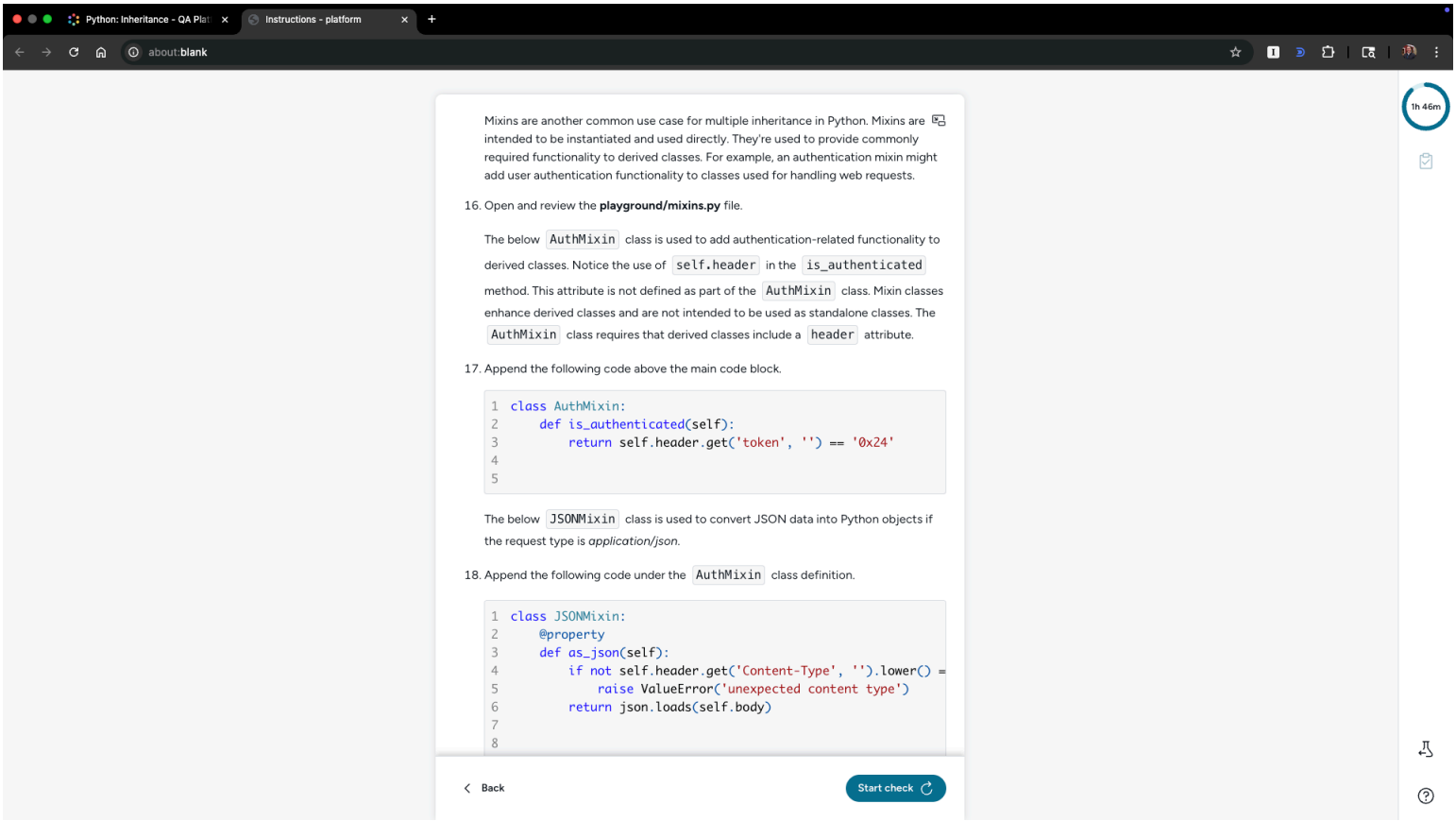
Notice that the keys are normalized from the `NormalizedDict` class. The values

Back

Start check

1h 47m


Python: Inheritance - QA Plat
Instructions - platform
about:blank
1h 47m
13. Run the code to observe the results.
PlayerScoreDict(<class 'int'>, {'adalovelace': 1314000, 'carlsagan': 1236000, 'gracehopper': 2349000})
The value for MISSING is: 0
Notice that the keys are normalized from the NormalizedDict class. The values are adjusted by a factor of 1_000 from the AdjustedValueDict class. The defaultdict ensures that using the dictionary lookup syntax for the key MISSING returns a default value rather than raising an exception.
The PlayerScoreDict class takes on the functionality from each base class. This is possible due to using super for method delagation.
14. Append the following code to the main code block to observe the mro for the PlayerScoreDict class.
1 print(PlayerScoreDict.mro())
15. Run the code to observe the results.
Because each base class has the same parentage the method resolution order matches the order that base classes are defined. Each base class delegates its calls to its base class. This ensures that the functionality of each base class is applied before delegating to the next base class.
Using multiple inheritance as a method delagation mechanism can produce more maintainable and reusable code. This design style requires careful consideration for method signatures. Derived classes must be able to provide the required arguments to multiple base classes.
Mixins are another common use case for multiple inheritance in Python. Mixins are not intended to be instantiated and used directly. They're used to provide commonly required functionality to derived classes. For example, an authentication mixin might add user authentication functionality to classes used for handling web requests.
16. Open and review the playground/mixins.py file.
Back
Start check


Python: Inheritance - QA Plat
Instructions - platform
about:blank
Mixins are another common use case for multiple inheritance in Python. Mixins are intended to be instantiated and used directly. They're used to provide commonly required functionality to derived classes. For example, an authentication mixin might add user authentication functionality to classes used for handling web requests.
16. Open and review the **playground/mixins.py** file.
The below `AuthMixin` class is used to add authentication-related functionality to derived classes. Notice the use of `self.header` in the `is_authenticated` method. This attribute is not defined as part of the `AuthMixin` class. Mixin classes enhance derived classes and are not intended to be used as standalone classes. The `AuthMixin` class requires that derived classes include a `header` attribute.
17. Append the following code above the main code block.
class AuthMixin:
def is_authenticated(self):
return self.header.get('token', '') == '0x24'
The below `JSONMixin` class is used to convert JSON data into Python objects if the request type is *application/json*.
18. Append the following code under the `AuthMixin` class definition.
class JSONMixin:
@property
def as_json(self):
if not self.header.get('Content-Type', '').lower() =
raise ValueError('unexpected content type')
return json.loads(self.body)
Back
Start check
1h 46m

Python: Inheritance - QA Plat | Instructions - platform

about:blank

1h 46m

18. Append the following code under the `AuthMixin` class definition.

```
class JSONMixin:
    @property
    def as_json(self):
        if not self.header.get('Content-Type', '').lower() =
            raise ValueError('unexpected content type')
        return json.loads(self.body)


```

The below `Request` class inherits from both mixin classes. The `process` method uses mixin methods to perform authentication and deserialization.

19. Append the following code under the `JSONMixin` class definition.

```
class Request(AuthMixin, JSONMixin):
    def __init__(self, header, body):
        self.header = header
        self.body = body

    def process(self):
        if not self.is_authenticated():
            return 'invalid user'
        return self.as_json


```

```
import json

class AuthMixin:
    def is_authenticated(self):
        return self.header.get('token', '') == '0x24'

class JSONMixin:
    @property
    def as_json(self):
        if not self.header.get('Content-Type', '').lower() == 'application/json':
            raise ValueError('unexpected content type')
```

Back

Start check

Python: Inheritance - QA Plat | Instructions - platform

about:blank

1h 46m

```python
import json

class AuthMixin:
    def is_authenticated(self):
        return self.header.get('token', '') == '0x24'

class JSONMixin:
    @property
    def as_json(self):
        if not self.header.get('Content-Type', '').lower() == 'application/json':
            raise ValueError('unexpected content type')
        return json.loads(self.body)

class Request(AuthMixin, JSONMixin):
    def __init__(self, header, body):
        self.header = header
        self.body = body

    def process(self):
        if not self.is_authenticated():
            return 'invalid user'
        return self.as_json

if __name__ == '__main__':
    print(Request({'token': '0x24', 'Content-Type': 'application/json'}, '[0, 1, 2, 3, 5]').process())
    print(Request({'token': 'xxxx', 'Content-Type': 'application/json'}, '[6, 7, 8, 9, 5]').process())
```

20. Run the code to observe the results.

```
python3 playground/mixins.py
```

```
[0, 1, 2, 3, 5]
invalid user
```

Carefully designed mixins can prevent the risk of ambiguous method resolution by providing a limited number of very specific methods.

⭐ Proceed to the next step ⭐

Validations

Multiple Inheritance: On-Track

1. Ensure that you're on-track.

Back

Start check