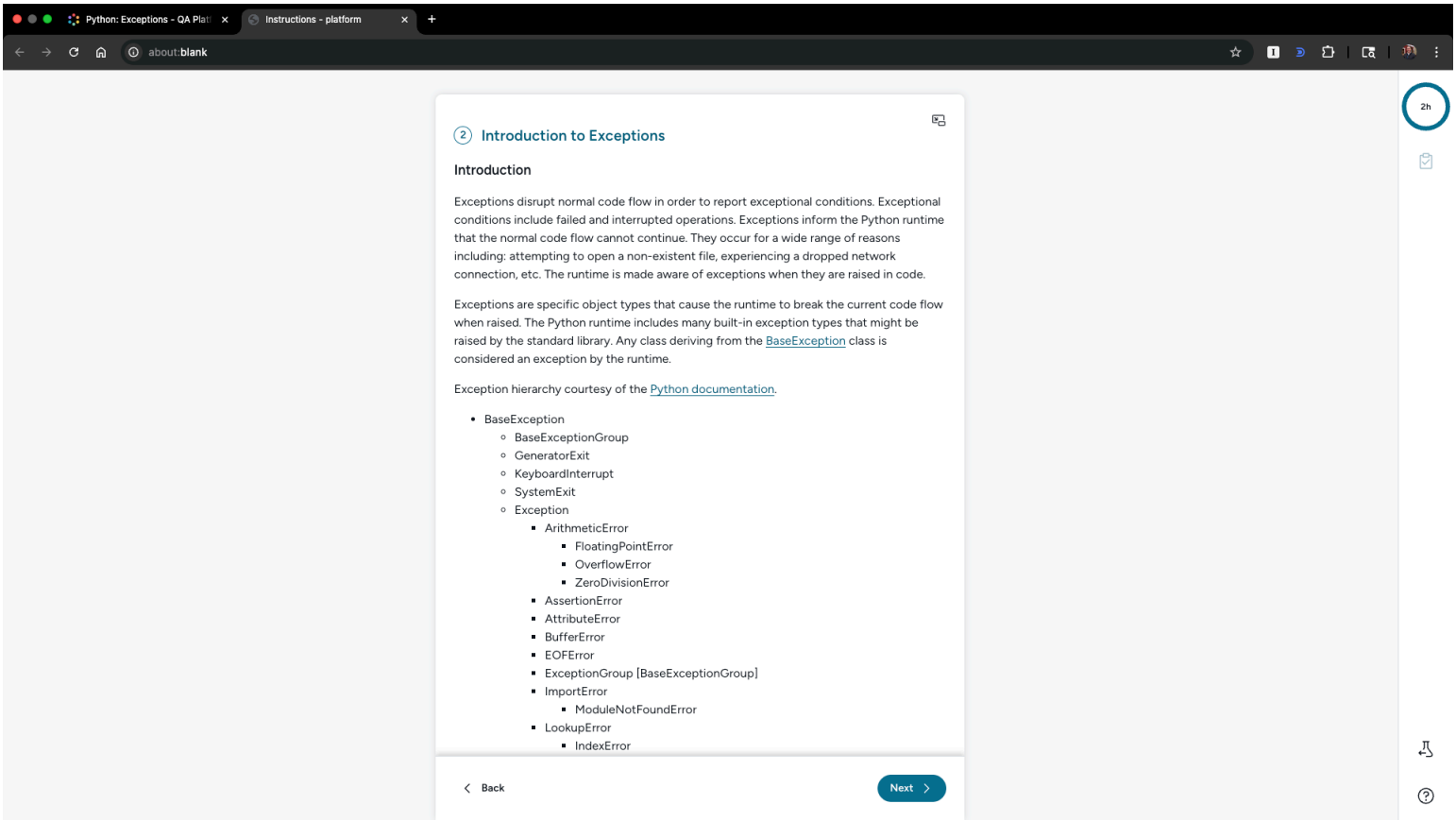
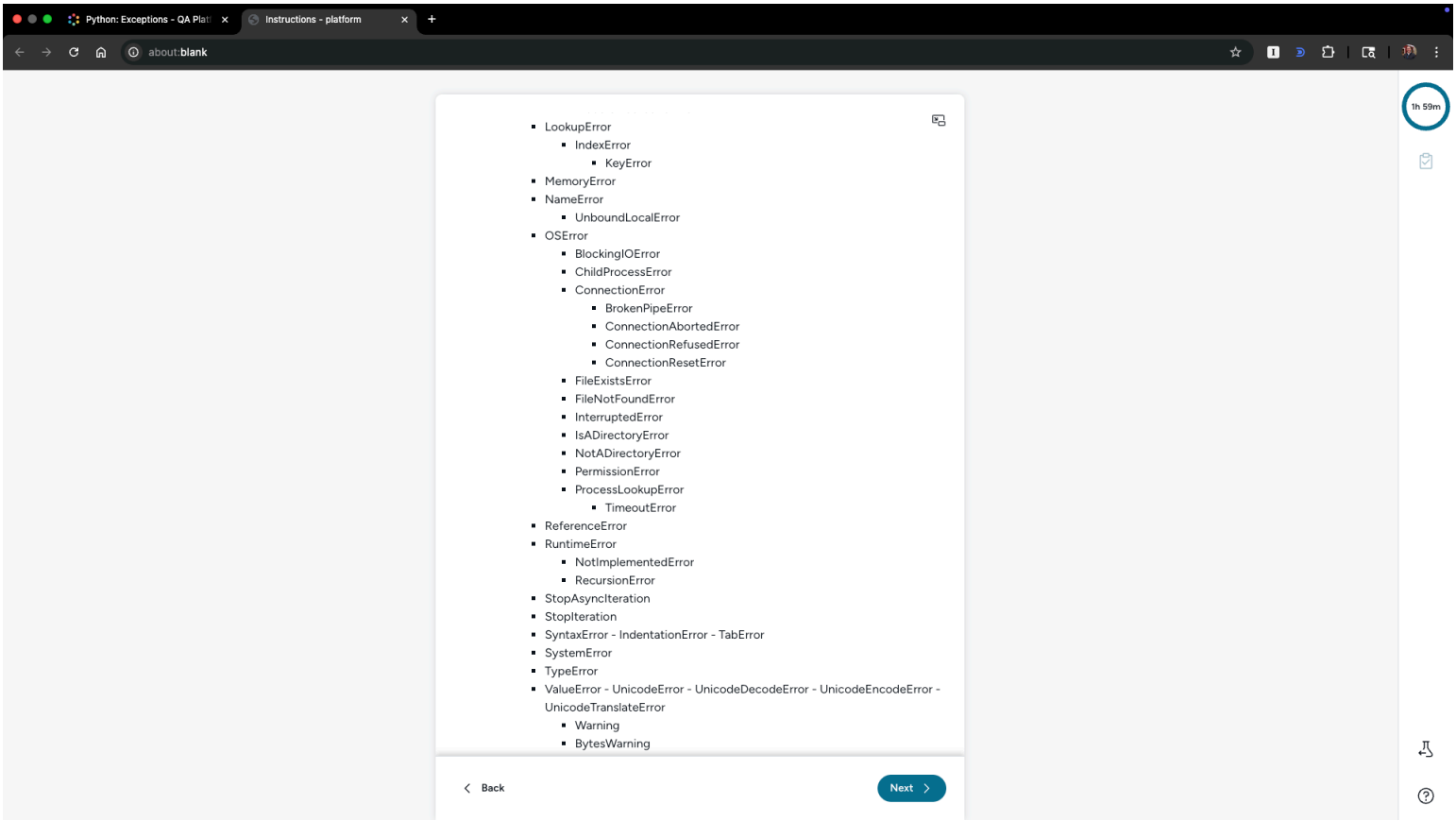

Python: Exceptions - QA Platf
Instructions - platform
about:blank
2h
2 Introduction to Exceptions
Introduction
Exceptions disrupt normal code flow in order to report exceptional conditions. Exceptional conditions include failed and interrupted operations. Exceptions inform the Python runtime that the normal code flow cannot continue. They occur for a wide range of reasons including: attempting to open a non-existent file, experiencing a dropped network connection, etc. The runtime is made aware of exceptions when they are raised in code.
Exceptions are specific object types that cause the runtime to break the current code flow when raised. The Python runtime includes many built-in exception types that might be raised by the standard library. Any class deriving from the BaseException class is considered an exception by the runtime.
Exception hierarchy courtesy of the Python documentation.
BaseException
BaseExceptionGroup
GeneratorExit
KeyboardInterrupt
SystemExit
Exception
ArithmeticError
FloatingPointError
OverflowError
ZeroDivisionError
AssertionError
AttributeError
BufferError
EOFError
ExceptionGroup [BaseExceptionGroup]
ImportError
ModuleNotFoundError
LookupError
IndexError
Back
Next

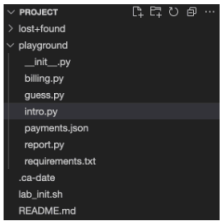

Python: Exceptions - QA Platf
Instructions - platform
about:blank
1h 59m
LookupError
IndexError
KeyError
MemoryError
NameError
UnboundLocalError
OSError
BlockingIOError
ChildProcessError
ConnectionError
BrokenPipeError
ConnectionAbortedError
ConnectionRefusedError
ConnectionResetError
FileExistsError
FileNotFoundError
InterruptedError
IsADirectoryError
NotADirectoryError
PermissionError
ProcessLookupError
TimeoutError
ReferenceError
RuntimeError
NotImplementedError
RecursionError
StopAsyncIteration
StopIteration
SyntaxError - IndentationError - TabError
SystemError
TypeError
ValueError - UnicodeError - UnicodeDecodeError - UnicodeEncodeError - UnicodeTranslateError
Warning
BytesWarning
Back
Next

- ValueError - UnicodeError - UnicodeDecodeError - UnicodeEncodeError
  UnicodeTranslateError
- Warning
- BytesWarning
- DeprecationWarning
- EncodingWarning
- FutureWarning
- ImportWarning
- PendingDeprecationWarning
- ResourceWarning
- RuntimeWarning
- SyntaxWarning
- UnicodeWarning
  - UserWarning

Each class in the above hierarchy represents a standard library exception. The runtime makes these exception types globally available allowing them to be used without requiring imports.

Instructions

1. Locate and open the **playground/intro.py** file using the **IDE's Explorer/project** pane.



Consider the following code example which attempts to call the built-in `print` callable without the required parentheses.

Python: Exceptions - QA Platf | Instructions - platform | about:blank

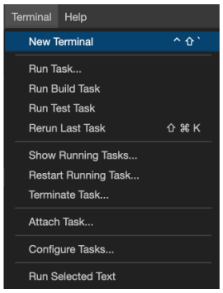
1h 59m

Consider the following code example which attempts to call the built-in `print` callable without the required parentheses.

2. Add the following code to the **playground/intro.py** file.

```
1 print 'hello'
```

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



4. Run the code in the terminal to observe the exception.

```
python3 playground/intro.py
```

```
  File "/home/project/playground/intro.py", line 1
    print 'hello'
          ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean print('hello')?
```

Python developers at all experience levels encounter syntax errors. Handling syntax errors is conceptually simple. Changing the code to follow Python's syntax rules resolves the exception. Due to the singular solution for syntax related exceptions no further exploration is required.

Back | Next

Non-syntax related exceptions are raised when an operation cannot or should not continue. The following code causes a `ZeroDivisionError` exception to be raised by performing the invalid operation of dividing a number by zero.

5. Replace the code in the **playground/intro.py** file with the following.

```
print(100 / 0)
```

6. Run the code to observe the exception.

```
python3 playground/intro.py
```

```
Traceback (most recent call last):
  File "/home/project/playground/intro.py", line 1, in <module>
    print(100 / 0)
ZeroDivisionError: division by zero
```

The runtime writes the details of unhandled exceptions to the console's stderr by default. The last line of output displays which exception occurred. The lines above display the location of the exception and a stack traceback. Tracebacks provide details about the state of the call stack when the exception occurred. These include the series of calls leading up to the exception and can become quite verbose.

Raising exceptions is accomplished by using the keyword `raise` followed by either an exception **type** or **instance**.

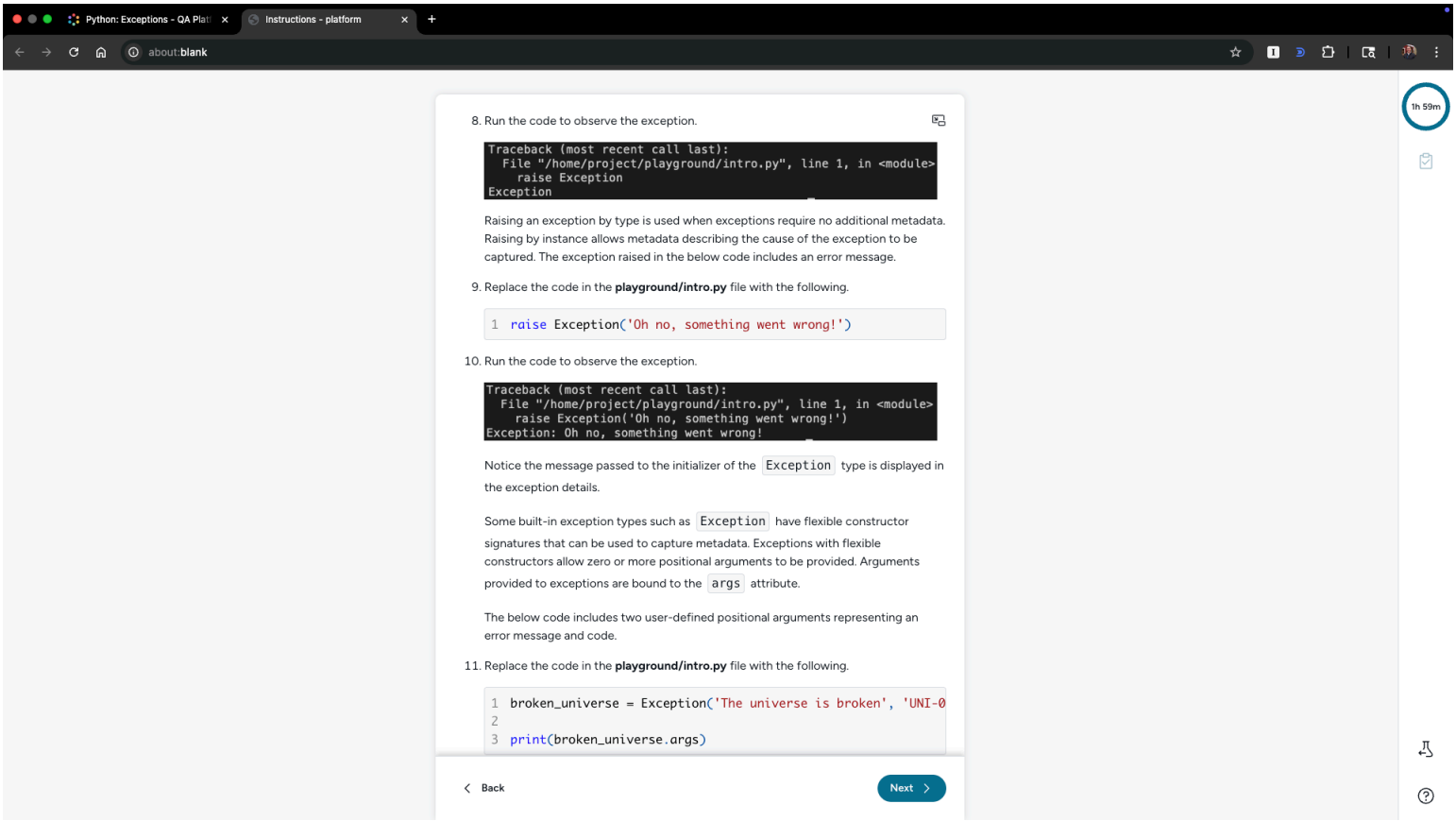
The following demonstrates raising an exception using an exception type.

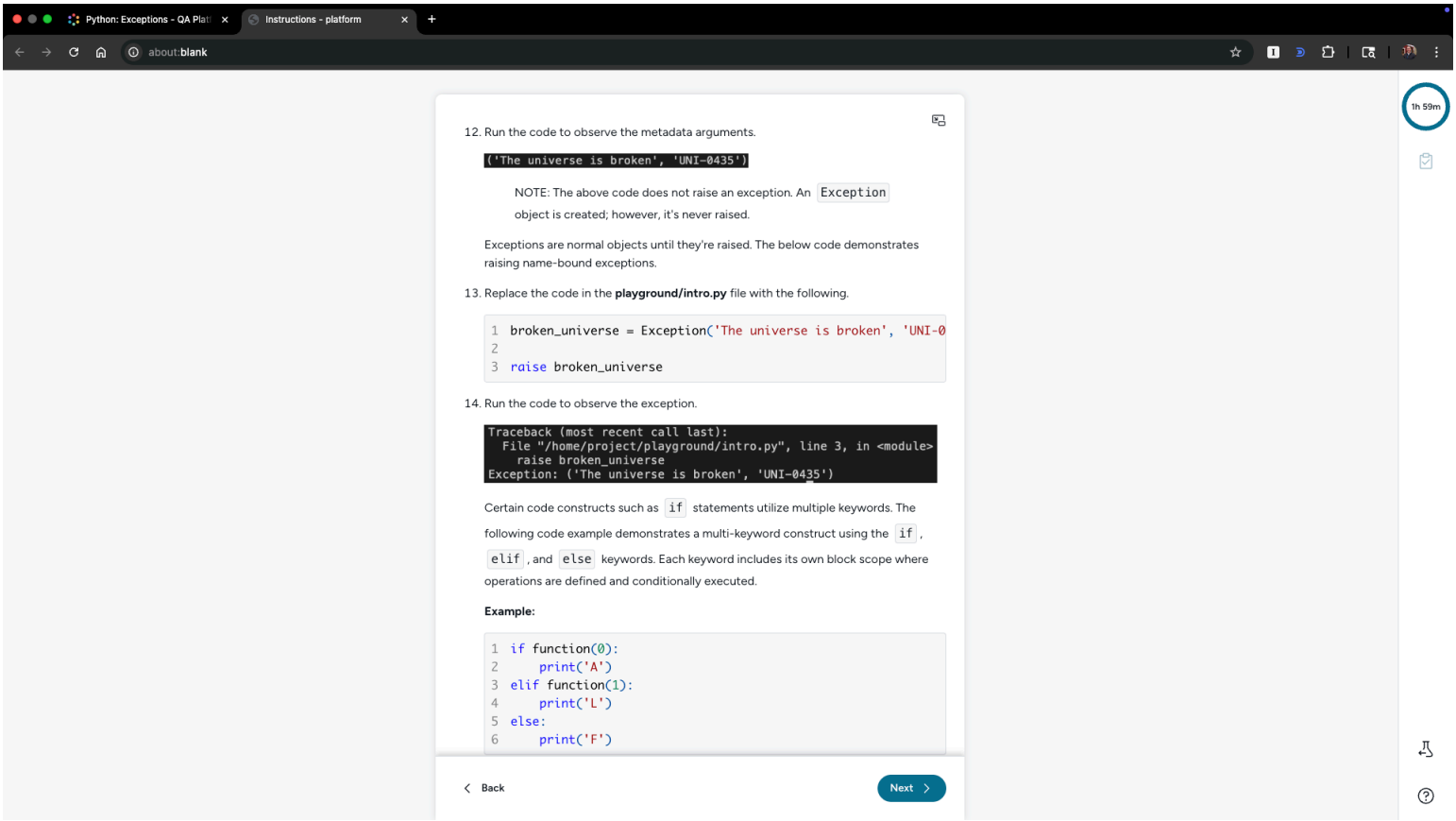
7. Replace the code in the **playground/intro.py** file with the following.

```
raise Exception
```

8. Run the code to observe the exception.

```
Traceback (most recent call last):
  File "/home/project/playground/intro.py", line 1, in <module>
    raise Exception
Exception
```


Python: Exceptions - QA Platf
Instructions - platform
about:blank
1h 59m
8. Run the code to observe the exception.
Traceback (most recent call last):
File "/home/project/playground/intro.py", line 1, in <module>
raise Exception
Exception
Raising an exception by type is used when exceptions require no additional metadata. Raising by instance allows metadata describing the cause of the exception to be captured. The exception raised in the below code includes an error message.
9. Replace the code in the playground/intro.py file with the following.
1 raise Exception('Oh no, something went wrong!')
10. Run the code to observe the exception.
Traceback (most recent call last):
File "/home/project/playground/intro.py", line 1, in <module>
raise Exception('Oh no, something went wrong!')
Exception: Oh no, something went wrong!
Notice the message passed to the initializer of the Exception type is displayed in the exception details.
Some built-in exception types such as Exception have flexible constructor signatures that can be used to capture metadata. Exceptions with flexible constructors allow zero or more positional arguments to be provided. Arguments provided to exceptions are bound to the args attribute.
The below code includes two user-defined positional arguments representing an error message and code.
11. Replace the code in the playground/intro.py file with the following.
1 broken_universe = Exception('The universe is broken', 'UNI-0
2
3 print(broken_universe.args)
Back
Next


Python: Exceptions - QA Platf
Instructions - platform
about:blank
1h 59m
12. Run the code to observe the metadata arguments.
('The universe is broken', 'UNI-0435')
NOTE: The above code does not raise an exception. An Exception object is created; however, it's never raised.
Exceptions are normal objects until they're raised. The below code demonstrates raising name-bound exceptions.
13. Replace the code in the playground/intro.py file with the following.
1 broken_universe = Exception('The universe is broken', 'UNI-0
2
3 raise broken_universe
14. Run the code to observe the exception.
Traceback (most recent call last):
File "/home/project/playground/intro.py", line 3, in <module>
raise broken_universe
Exception: ('The universe is broken', 'UNI-0435')
Certain code constructs such as if statements utilize multiple keywords. The following code example demonstrates a multi-keyword construct using the if, elif, and else keywords. Each keyword includes its own block scope where operations are defined and conditionally executed.
Example:
1 if function(0):
2 print('A')
3 elif function(1):
4 print('L')
5 else:
6 print('F')
Back
Next

The exception handling construct known as a **try-except** statement uses multiple keywords. The keywords `try`, `except`, `else`, and `finally` are used as part of Python's exception handling code construct.

Try-except statements must define an opening and closing block. The opening block is always a `try` block however, the closing block is more flexible. The closing block can be an `except` and/or `finally` block.

Each of the keywords introduces its own block scope. The Python runtime includes logic that determines which blocks are run and when. Try blocks are always defined first and run until the end of the block is reached or an exception is raised. Except blocks are called only if an exception is raised inside the try block. Else blocks are called only if the `try` block runs without exception. Finally blocks are always run last no matter what occurs.

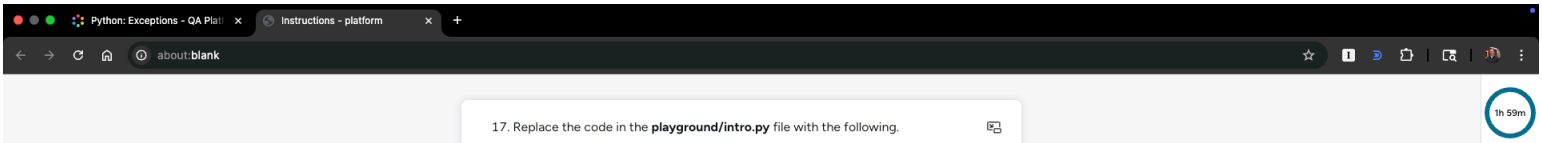
15. Replace the code in the **playground/intro.py** file with the following.

```
try:
    print('from try')
except:
    print('from except')
else:
    print('from else')
finally:
    print('from finally')

```

16. Run the code to observe the results.

```
from try
from else
from finally
```

Raising an exception inside the try block will result in the except block being run rather than the else block.



17. Replace the code in the **playground/intro.py** file with the following.

```
try:
    print('from try')
    raise Exception
except:
    print('from except')
else:
    print('from else')
finally:
    print('from finally')

```

```
from try
from except
from finally
```

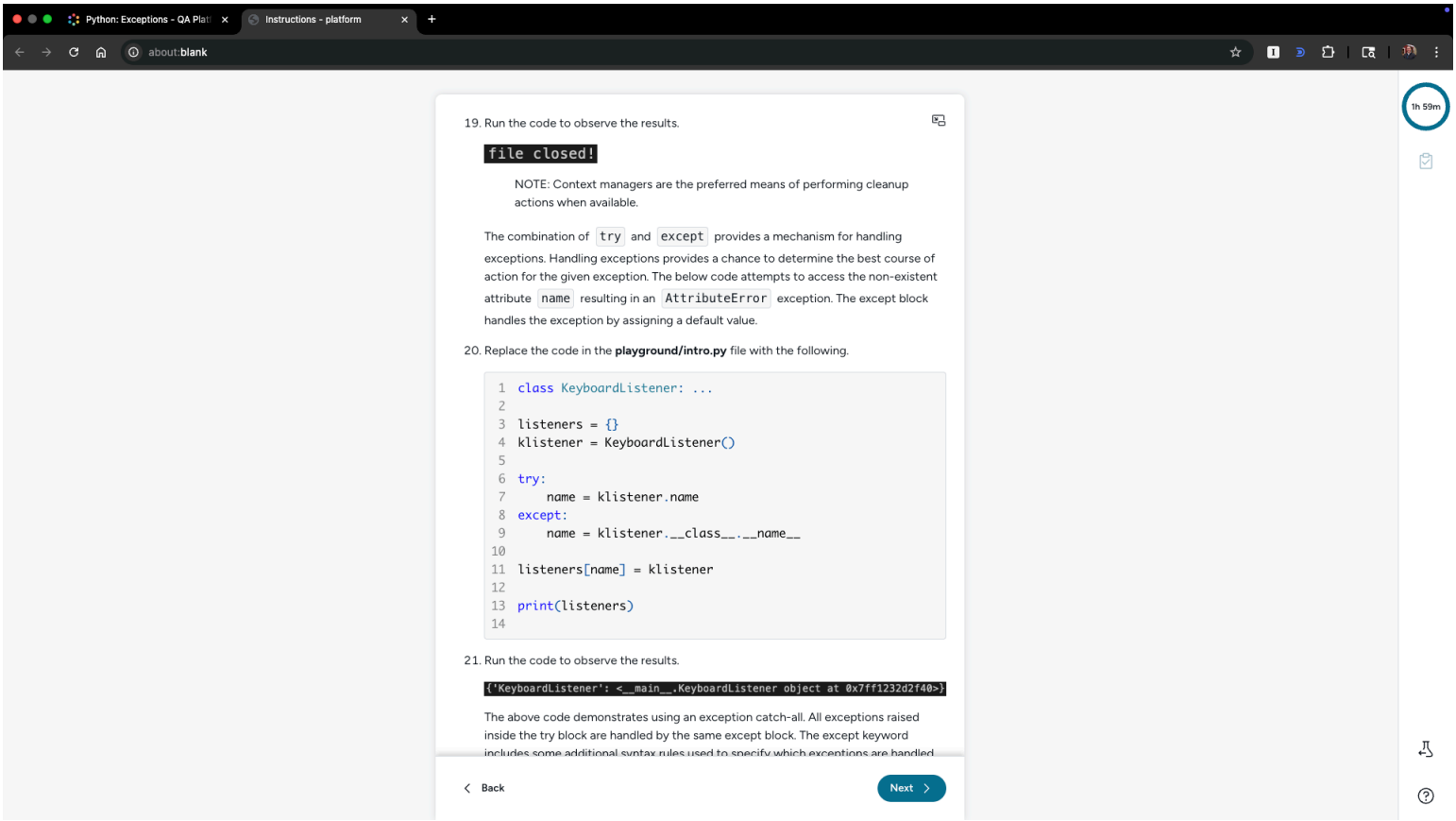
The combination of `try` and `finally` provides a mechanism for performing cleanup operations such as closing files, network and database connections, etc. The below code opens a file in write mode, ensuring it will be created if it doesn't already exist. After writing text to the file the try block completes and the finally block is run to close the file.

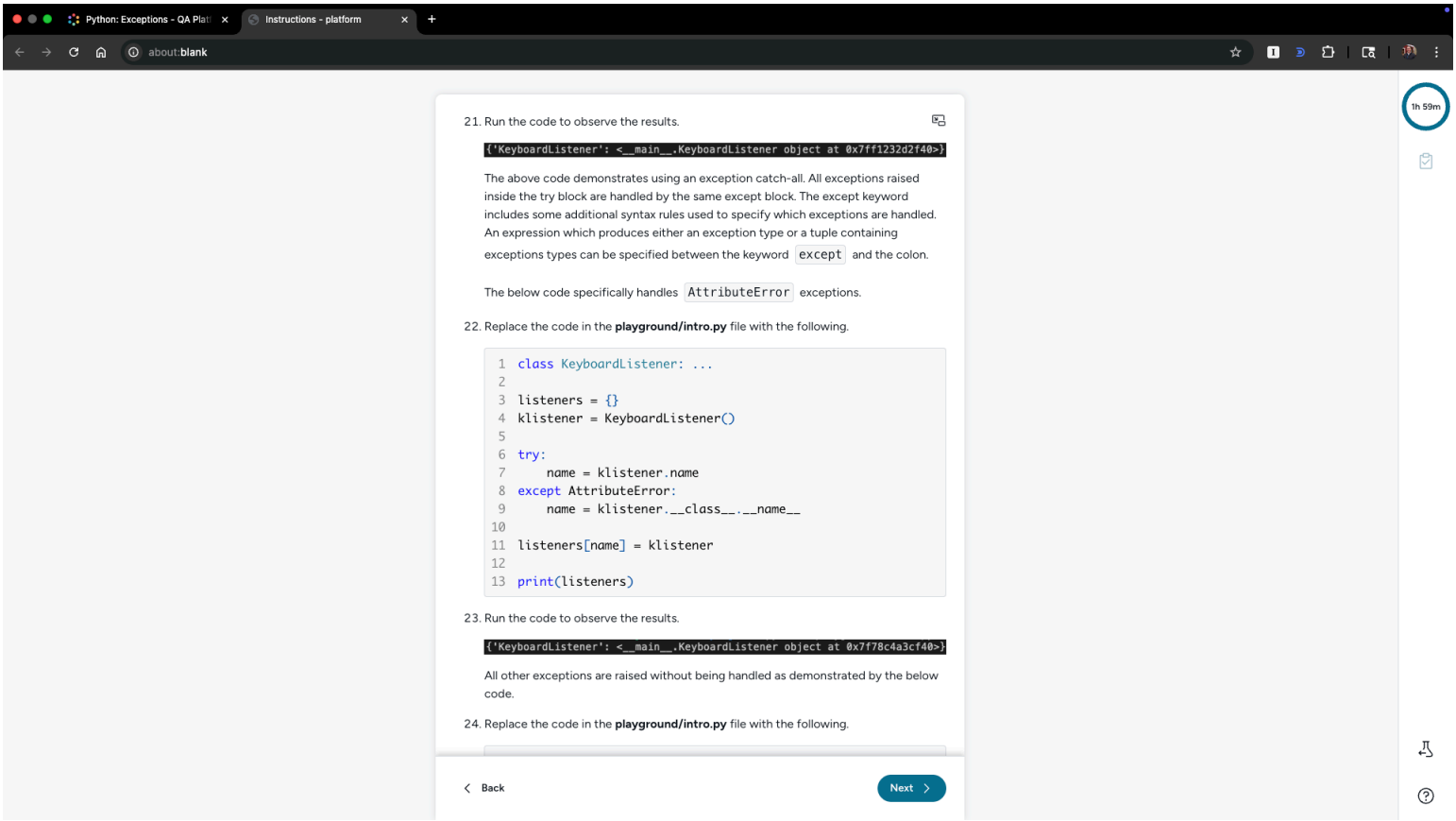
18. Replace the code in the **playground/intro.py** file with the following.

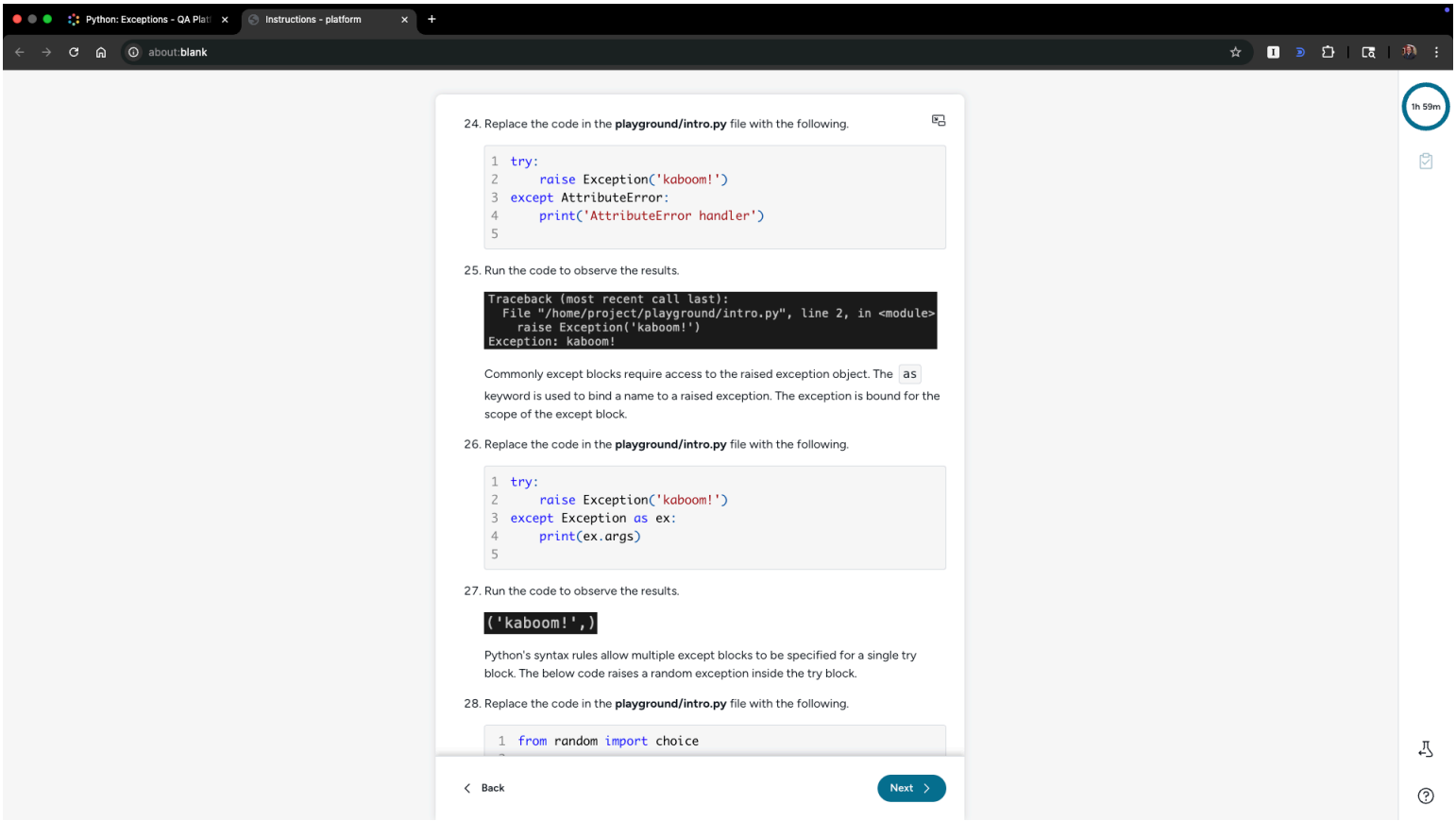
```
try:
    f = open('dataset.txt', 'w')
    f.write('python is neat')
finally:
    f.close()
    print('file closed!')

```

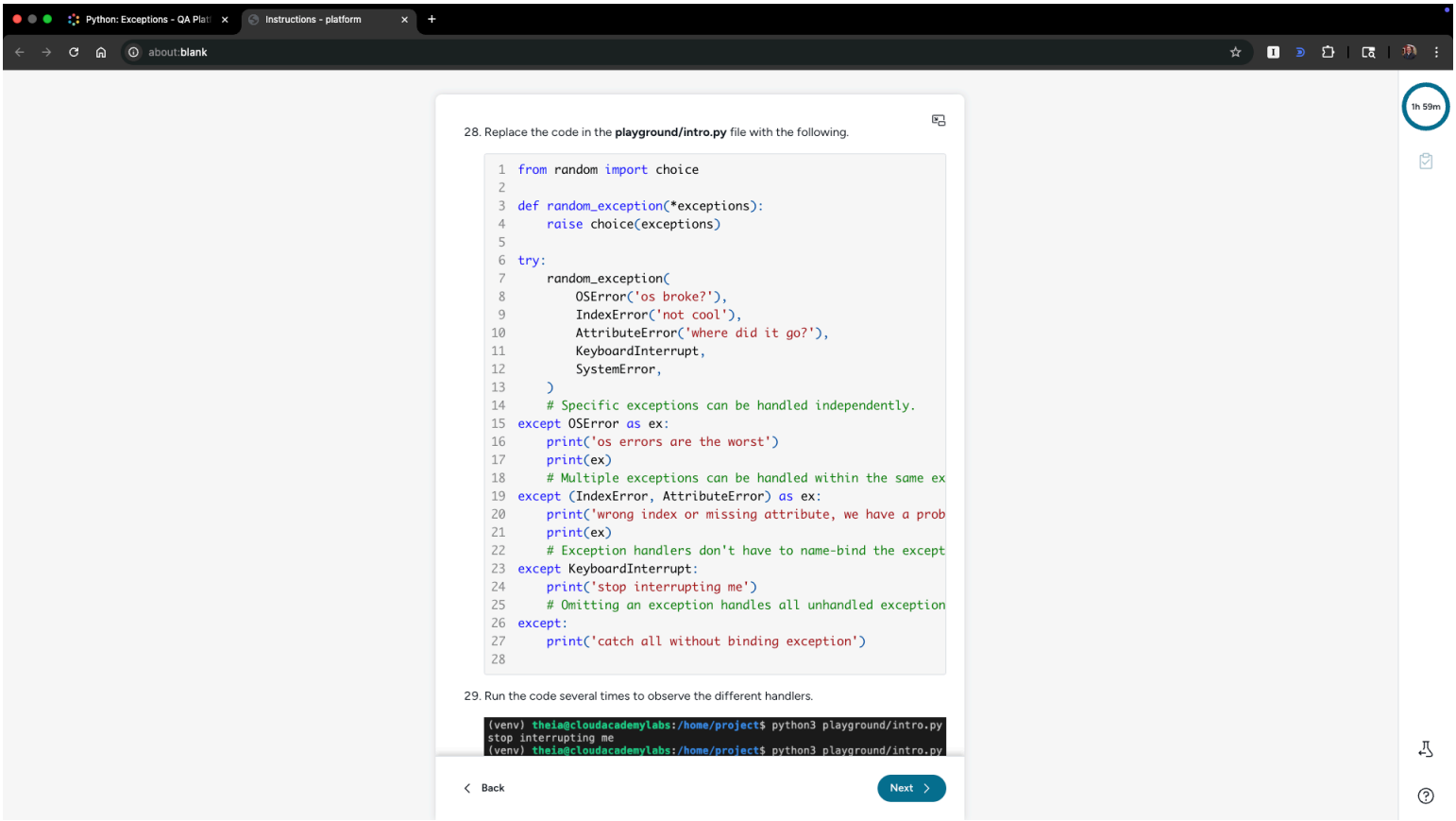
19. Run the code to observe the results.

```
file closed!
```


Python: Exceptions - QA Platf
Instructions - platform
about:blank
1h 59m
19. Run the code to observe the results.
file closed!
NOTE: Context managers are the preferred means of performing cleanup actions when available.
The combination of try and except provides a mechanism for handling exceptions. Handling exceptions provides a chance to determine the best course of action for the given exception. The below code attempts to access the non-existent attribute name resulting in an AttributeError exception. The except block handles the exception by assigning a default value.
20. Replace the code in the playground/intro.py file with the following.
class KeyboardListener: ...
listeners = {}
klistener = KeyboardListener()
try:
name = klistener.name
except:
name = klistener.__class__.__name__
listeners[name] = klistener
print(listeners)
21. Run the code to observe the results.
{'KeyboardListener': <__main__.KeyboardListener object at 0x7ff1232d2f40>}
The above code demonstrates using an exception catch-all. All exceptions raised inside the try block are handled by the same except block. The except keyword includes some additional syntax rules used to specify which exceptions are handled
Back
Next


Python: Exceptions - QA Platf
Instructions - platform
about:blank
1h 59m
21. Run the code to observe the results.
{'KeyboardListener': <__main__.KeyboardListener object at 0x7ff1232d2f40>}
The above code demonstrates using an exception catch-all. All exceptions raised inside the try block are handled by the same except block. The except keyword includes some additional syntax rules used to specify which exceptions are handled. An expression which produces either an exception type or a tuple containing exceptions types can be specified between the keyword except and the colon.
The below code specifically handles AttributeError exceptions.
22. Replace the code in the playground/intro.py file with the following.
class KeyboardListener: ...
listeners = {}
klistener = KeyboardListener()
try:
name = klistener.name
except AttributeError:
name = klistener.__class__.__name__
listeners[name] = klistener
print(listeners)
23. Run the code to observe the results.
{'KeyboardListener': <__main__.KeyboardListener object at 0x7f78c4a3cf40>}
All other exceptions are raised without being handled as demonstrated by the below code.
24. Replace the code in the playground/intro.py file with the following.
Back
Next


Python: Exceptions - QA Platf
Instructions - platform
about:blank
1h 59m
24. Replace the code in the playground/intro.py file with the following.
try:
raise Exception('kaboom!')
except AttributeError:
print('AttributeError handler')
25. Run the code to observe the results.
Traceback (most recent call last):
File "/home/project/playground/intro.py", line 2, in <module>
raise Exception('kaboom!')
Exception: kaboom!
Commonly except blocks require access to the raised exception object. The as keyword is used to bind a name to a raised exception. The exception is bound for the scope of the except block.
26. Replace the code in the playground/intro.py file with the following.
try:
raise Exception('kaboom!')
except Exception as ex:
print(ex.args)
27. Run the code to observe the results.
('kaboom!',)
Python's syntax rules allow multiple except blocks to be specified for a single try block. The below code raises a random exception inside the try block.
28. Replace the code in the playground/intro.py file with the following.
from random import choice
Back
Next


Python: Exceptions - QA Plat
Instructions - platform
about:blank
1h 59m
28. Replace the code in the playground/intro.py file with the following.
from random import choice
def random_exception(*exceptions):
raise choice(exceptions)
try:
random_exception(
OSError('os broke?'),
IndexError('not cool'),
AttributeError('where did it go?'),
KeyboardInterrupt,
SystemError,
)
Specific exceptions can be handled independently.
except OSError as ex:
print('os errors are the worst')
print(ex)
# Multiple exceptions can be handled within the same ex
except (IndexError, AttributeError) as ex:
print('wrong index or missing attribute, we have a prob
print(ex)
# Exception handlers don't have to name-bind the except
except KeyboardInterrupt:
print('stop interrupting me')
# Omitting an exception handles all unhandled exception
except:
print('catch all without binding exception')
29. Run the code several times to observe the different handlers.
(venv) theia@cloudacademylabs:/home/project$ python3 playground/intro.py
stop interrupting me
(venv) theia@cloudacademylabs:/home/project$ python3 playground/intro.py
Back
Next

29. Run the code several times to observe the different handlers.

```
(venv) theia@cloudacademylabs:/home/project$ python3 playground/intro.py
stop interrupting me
(venv) theia@cloudacademylabs:/home/project$ python3 playground/intro.py
os errors are the worst
os broke?
(venv) theia@cloudacademylabs:/home/project$ python3 playground/intro.py
wrong index or missing attribute, we have a problem.
where did it go?
(venv) theia@cloudacademylabs:/home/project$ python3 playground/intro.py
wrong index or missing attribute, we have a problem.
where did it go?
(venv) theia@cloudacademylabs:/home/project$ python3 playground/intro.py
stop interrupting me
(venv) theia@cloudacademylabs:/home/project$ python3 playground/intro.py
wrong index or missing attribute, we have a problem.
not cool
(venv) theia@cloudacademylabs:/home/project$ python3 playground/intro.py
catch all without binding exception
```

Exceptions are handled by the first capable `except` block. Both of the `except` blocks in the below code include `AttributeError` exceptions. However, only the first `except` block is run.

30. Replace the code in the **playground/intro.py** file with the following.

```
try:
    ':-)'.non_existent_attr()
except AttributeError as ex:
    print('missing attribute')
    print(ex)
except (IndexError, AttributeError) as ex:
    print('wrong index or missing attribute, we have a probl
    print(ex)

```

31. Run the code to observe the results.

```
missing attribute
'str' object has no attribute 'non_existent_attr'
```

31. Run the code to observe the results.

```
missing attribute
'str' object has no attribute 'non_existent_attr'
```

By specifying a base exception class as the exception to handle all derived types are also handled. Most built-in and custom exceptions derive from the `Exception` type. The below code defines an `Exception` handler as the first `except` block. The raised `AttributeError` is handled by the `Exception` handler because `AttributeError` derives from the `Exception` type.

32. Replace the code in the **playground/intro.py** file with the following.

```
try:
    ':-)'.non_existent_attr()
except Exception as ex:
    print('handling exceptions')
    print(ex)
except AttributeError as ex:
    print('missing attribute')
    print(ex)
except (IndexError, AttributeError) as ex:
    print('wrong index or missing attribute, we have a prob
    print(ex)

```

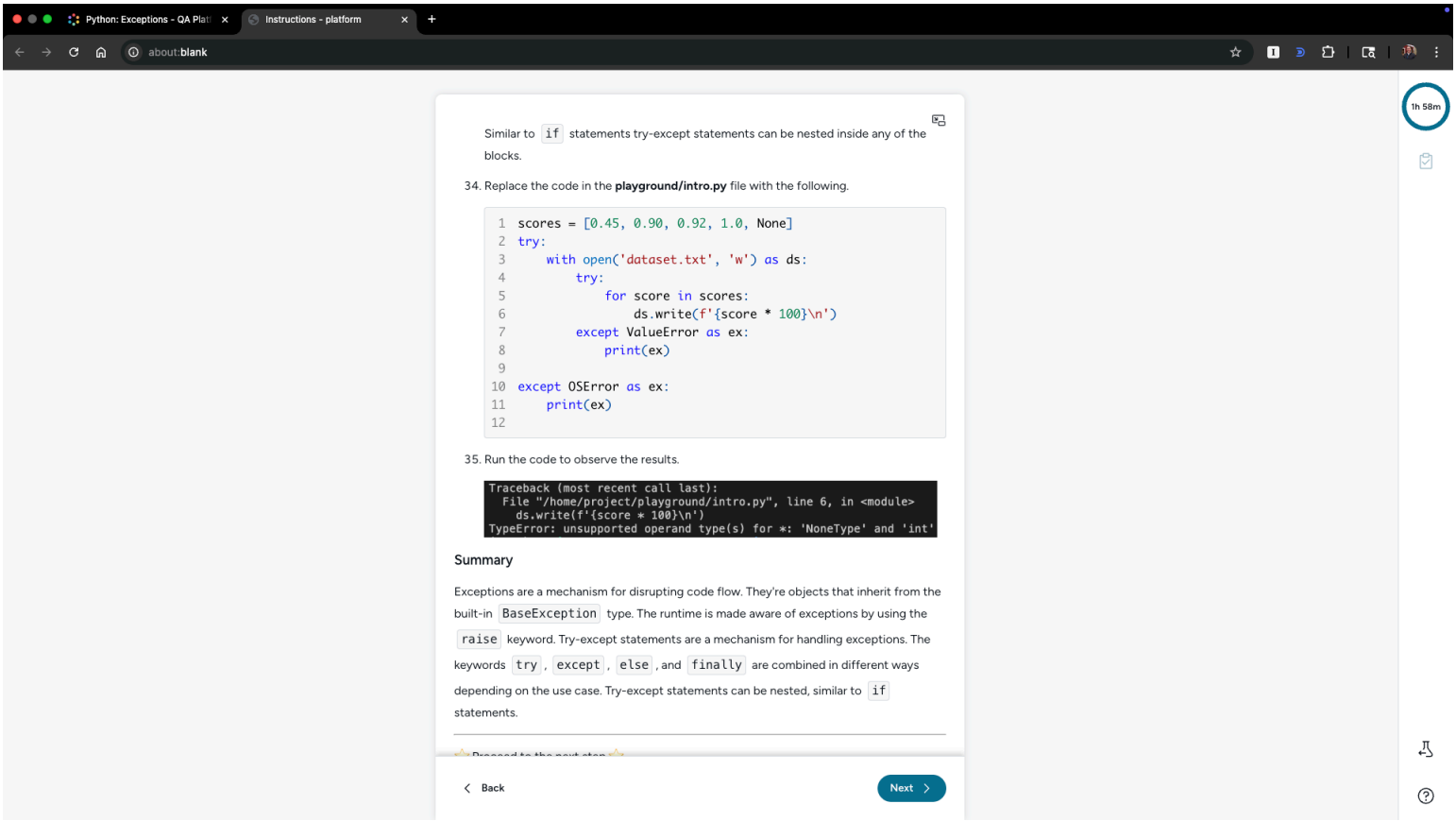
33. Run the code to observe the results.

```
handling exceptions
'str' object has no attribute 'non_existent_attr'
```

When handling exceptions it's important to consider the order of the handlers and the exception's class hierarchy. Placing a base class above a more specific exception will match with the base class.

Similar to `if` statements try-except statements can be nested inside any of the

Back
Next



Similar to `if` statements try-except statements can be nested inside any of the blocks.

34. Replace the code in the **playground/intro.py** file with the following.

```
scores = [0.45, 0.90, 0.92, 1.0, None]
try:
    with open('dataset.txt', 'w') as ds:
        try:
            for score in scores:
                ds.write(f'{score * 100}\n')
        except ValueError as ex:
            print(ex)

except OSError as ex:
    print(ex)

```

35. Run the code to observe the results.

```
Traceback (most recent call last):
  File "/home/project/playground/intro.py", line 6, in <module>
    ds.write(f'{score * 100}\n')
TypeError: unsupported operand type(s) for *: 'NoneType' and 'int'
```

Summary

Exceptions are a mechanism for disrupting code flow. They're objects that inherit from the built-in `BaseException` type. The runtime is made aware of exceptions by using the `raise` keyword. Try-except statements are a mechanism for handling exceptions. The keywords `try`, `except`, `else`, and `finally` are combined in different ways depending on the use case. Try-except statements can be nested, similar to `if` statements.

Proceed to the next step

Back

Next