Iterators and Generators

Objectives:

- Define iterator and iterable
- Understand the iter() and next() methods
- Build our own for loop
- Define what generators are and how then can be used
- Compare generator functions and generator expressions
- Use generators to pause execution of expensive functions

Iterators vs Iterables:

- **Iterator**—an object that can be iterated upon. An object which returns data, one element at a time when next() is called on it.
- **Iterable—**an object which will return an iterator when iter() is called on

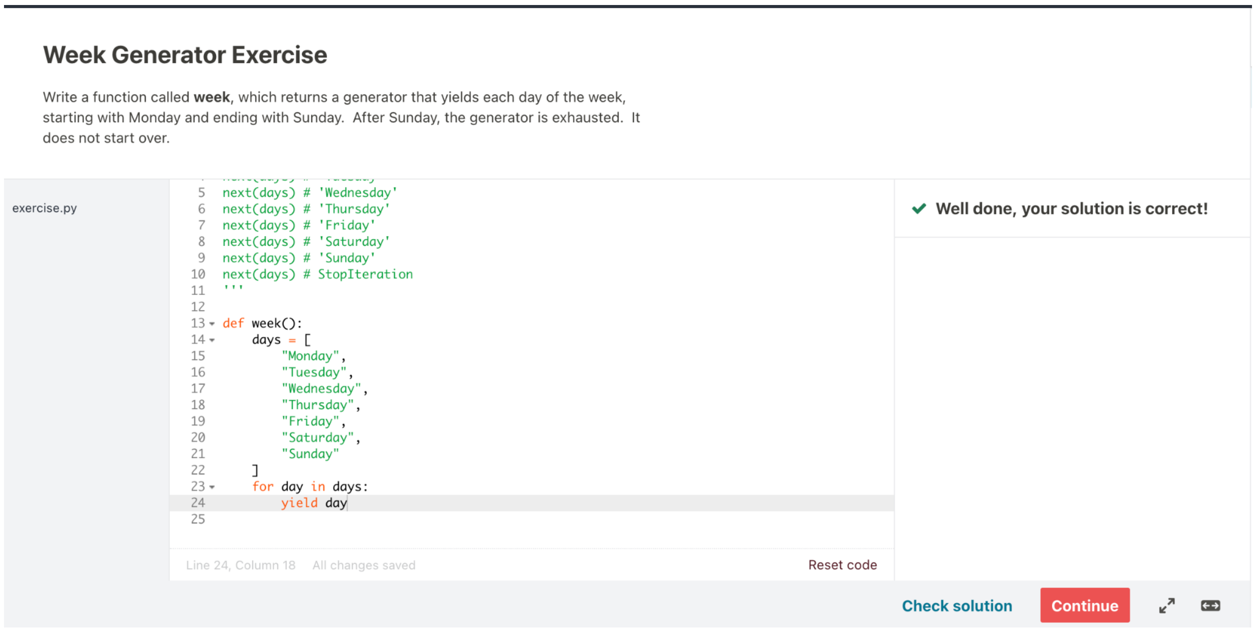
Writing a custom iterator i.e.:

```
class Counter:
    def __init__(self, low, high):
        self.current = low
        self.high = high

    def __iter__(self):
        return self

    def __next__(self):
        if self.current < self.high:
            num = self.current
            self.current += 1
            return num
        raise StopIteration

for x in Counter(50,70):
    print(x)
```

Generators:

- Generators are iterators
- Generators can be created with generator functions
- Generator functions can use the yield keyword
- Generators can be created with generator expressions

Functions	Generator Functions
uses return	uses yield
returns once	can yield multiple times
when invoked, returns the return value	when invoked, returns a generator



Week Generator Exercise

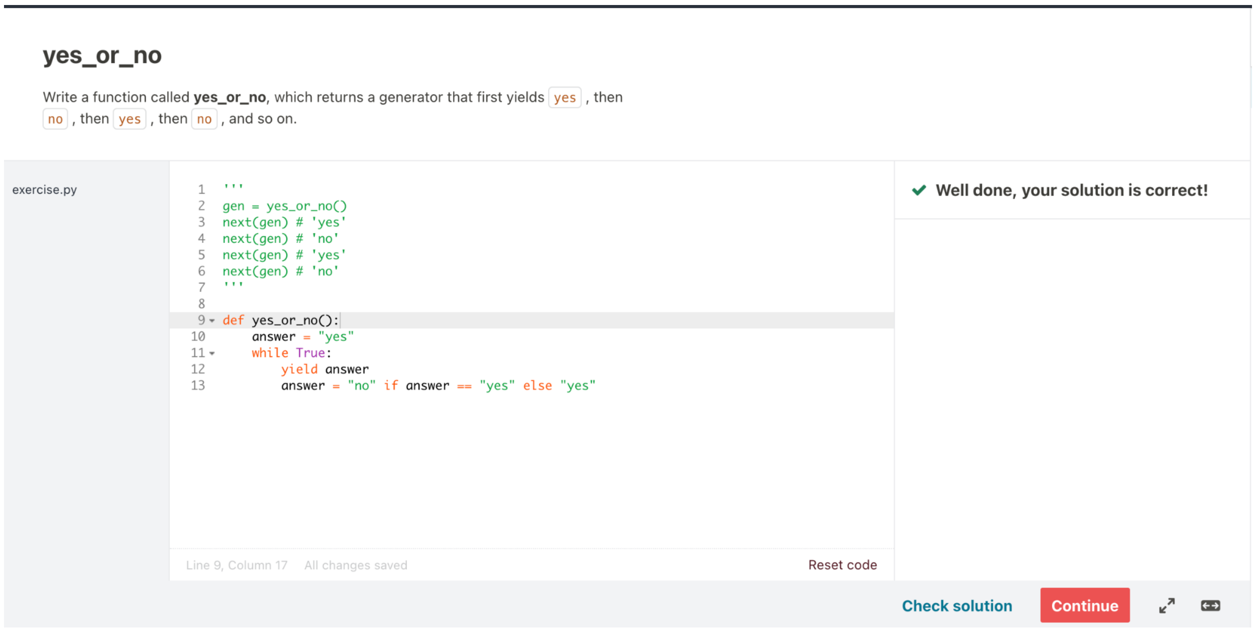
Write a function called **week**, which returns a generator that yields each day of the week, starting with Monday and ending with Sunday. After Sunday, the generator is exhausted. It does not start over.

exercise.py

```python
next(days) # 'Wednesday'
next(days) # 'Thursday'
next(days) # 'Friday'
next(days) # 'Saturday'
next(days) # 'Sunday'
next(days) # StopIteration
'''

def week():
    days = [
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday",
        "Saturday",
        "Sunday"
    ]
    for day in days:
        yield day
```

✓ Well done, your solution is correct!

Line 24, Column 18 All changes saved
Reset code
Check solution
Continue



yes_or_no

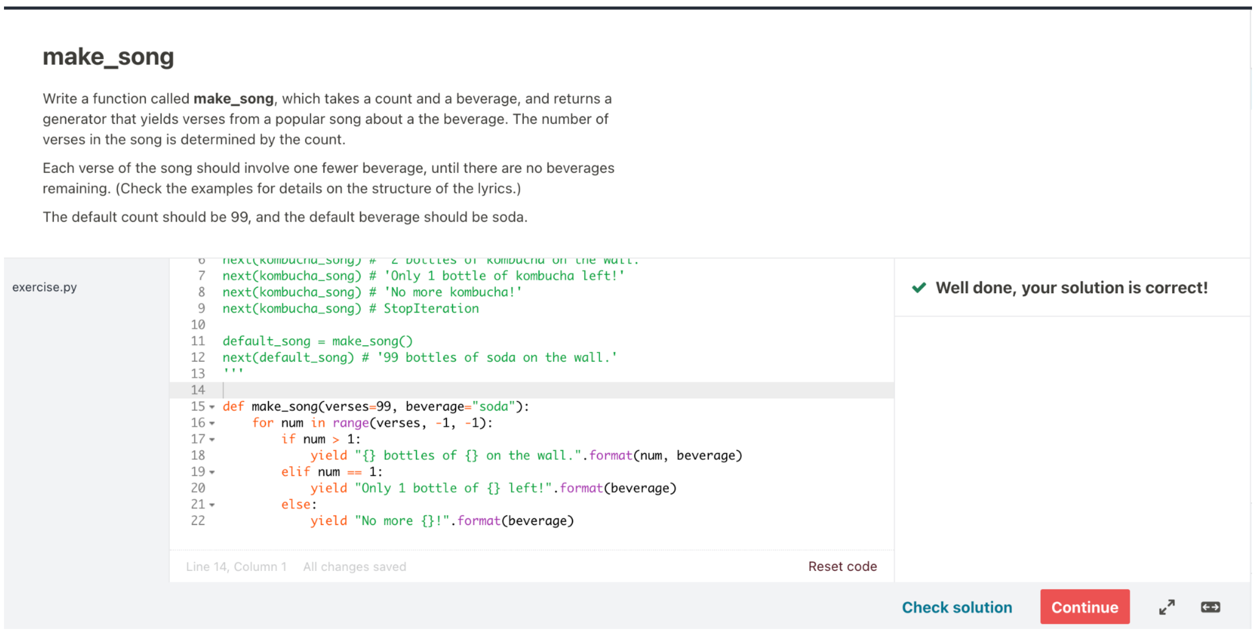
Write a function called **yes_or_no**, which returns a generator that first yields `yes` , then `no` , then `yes` , then `no` , and so on.

exercise.py

```python
'''
gen = yes_or_no()
next(gen) # 'yes'
next(gen) # 'no'
next(gen) # 'yes'
next(gen) # 'no'
'''

def yes_or_no():
    answer = "yes"
    while True:
        yield answer
        answer = "no" if answer == "yes" else "yes"
```

✓ Well done, your solution is correct!

Line 9, Column 17 All changes saved
Reset code
Check solution
Continue



make_song

Write a function called **make_song**, which takes a count and a beverage, and returns a generator that yields verses from a popular song about a the beverage. The number of verses in the song is determined by the count.

Each verse of the song should involve one fewer beverage, until there are no beverages remaining. (Check the examples for details on the structure of the lyrics.)

The default count should be 99, and the default beverage should be soda.

exercise.py

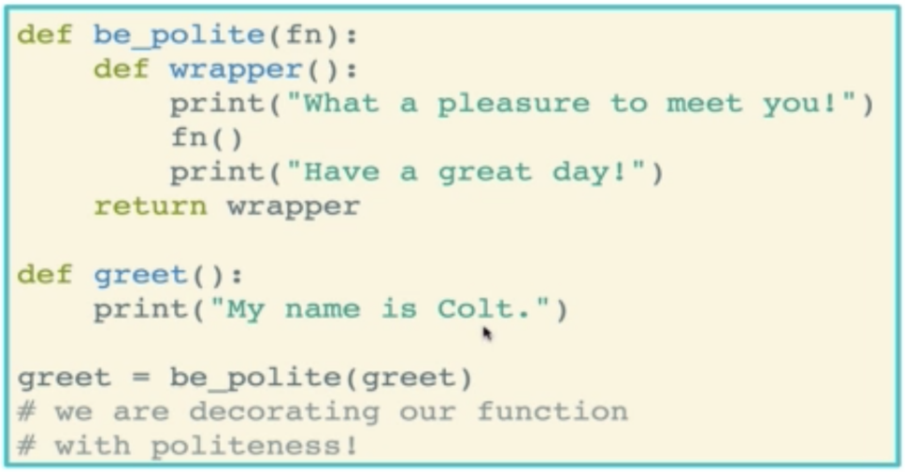
```python
next(kombucha_song) # '2 bottles of kombucha on the wall.'
next(kombucha_song) # 'Only 1 bottle of kombucha left!'
next(kombucha_song) # 'No more kombucha!'
next(kombucha_song) # StopIteration

default_song = make_song()
next(default_song) # '99 bottles of soda on the wall.'
'''

def make_song(verses=99, beverage="soda"):
    for num in range(verses, -1, -1):
        if num > 1:
            yield "{} bottles of {} on the wall.".format(num, beverage)
        elif num == 1:
            yield "Only 1 bottle of {} left!".format(beverage)
        else:
            yield "No more {}!".format(beverage)
```

Decorators

Higher Order Functions:
https://www.geeksforgeeks.org/higher-order-functions-in-python/

What's a Decorator?

- Decorators are functions
- Decorators wrap other functions and enhance their behavior
- Decorators are examples of higher order functions
- Decorators have their own syntax using "@" (syntactic sugar)



```python
def be_polite(fn):
    def wrapper():
        print("What a pleasure to meet you!")
        fn()
        print("Have a great day!")
    return wrapper

def greet():
    print("My name is Colt.")

greet = be_polite(greet)
# we are decorating our function
# with politeness!
```

```python
def be_polite(fn):
    def wrapper():
        print("What a pleasure to meet you!")
        fn()
        print("Have a great day!")
    return wrapper

@be_polite
def greet():
    print("My name is Matt.")

# we don't need to set
# greet = be_polite(greet)
```

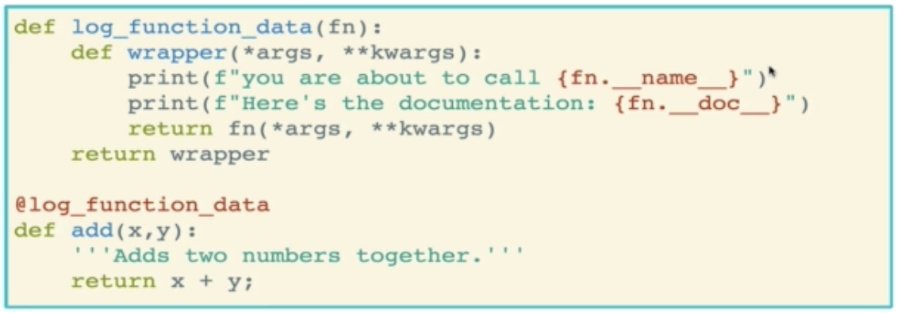
Functions with Different Signatures

```python
def shout(fn):
    def wrapper(name):
        return fn(name).upper()
    return wrapper

@shout
def greet(name):
    return f"Hi, I'm {name}."

@shout
def order(main, side):
    return f"Hi, I'd like the {main}, with a side of {side}, please."
```

Decorator Pattern

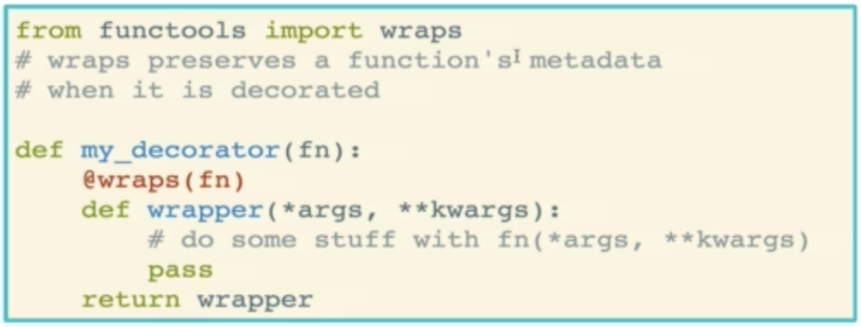
```python
def my_decorator(fn):
    def wrapper(*args, **kwargs):
        # do some stuff with fn(*args, **kwargs)
        pass
    return wrapper
```

Preserving Metadata



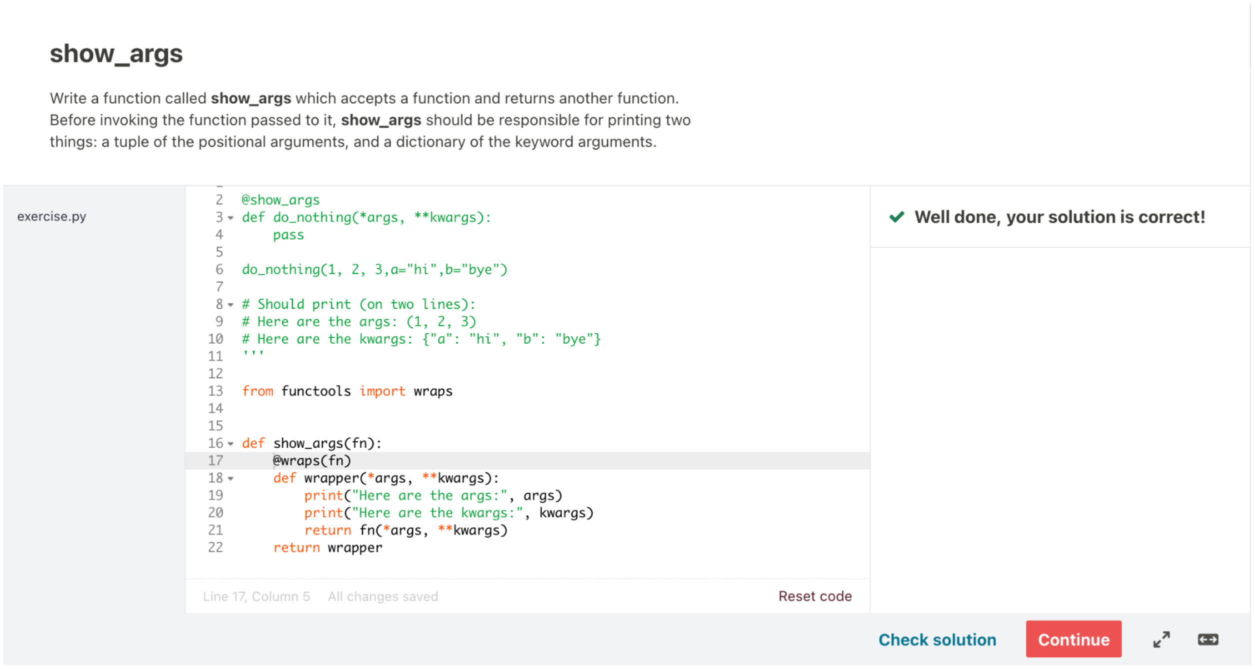
```python
def log_function_data(fn):
    def wrapper(*args, **kwargs):
        print(f"you are about to call {fn.__name__}")
        print(f"Here's the documentation: {fn.__doc__}")
        return fn(*args, **kwargs)
    return wrapper

@log_function_data
def add(x,y):
    '''Adds two numbers together.'''
    return x + y;
```

Decorator Pattern



```python
from functools import wraps
# wraps preserves a function's metadata
# when it is decorated

def my_decorator(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        # do some stuff with fn(*args, **kwargs)
        pass
    return wrapper
```

By using the functools wraps function, it makes sure that all the attributes in the function that we’re decorating aren’t lost by the decorator. It ensures that the metadata is not lost.



show_args

Write a function called **show_args** which accepts a function and returns another function. Before invoking the function passed to it, **show_args** should be responsible for printing two things: a tuple of the positional arguments, and a dictionary of the keyword arguments.

exercise.py

```python
@show_args
def do_nothing(*args, **kwargs):
    pass

do_nothing(1, 2, 3,a="hi",b="bye")

# Should print (on two lines):
# Here are the args: (1, 2, 3)
# Here are the kwargs: {"a": "hi", "b": "bye"}
'''


from functools import wraps


def show_args(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        print("Here are the args:", args)
        print("Here are the kwargs:", kwargs)
        return fn(*args, **kwargs)
    return wrapper
```

Line 17, Column 5 All changes saved

✓ Well done, your solution is correct!

Testing with Python

Objectives:

- Describe what tests are and why they are essential
- Explain what Test Driven Development is
- Test Python code using doctests
- Test Python code using assert
- Explain what unit testing is
- Write unit tests using the unittest module
- Remove code duplication using before and after hooks

Why test?

- Reduce bugs in existing code
- Ensure bugs that are fixed stay fixed
- Ensure that new features don’t break old ones
- Ensure that cleaning up code doesn’t introduce new bugs
- Makes development more fun

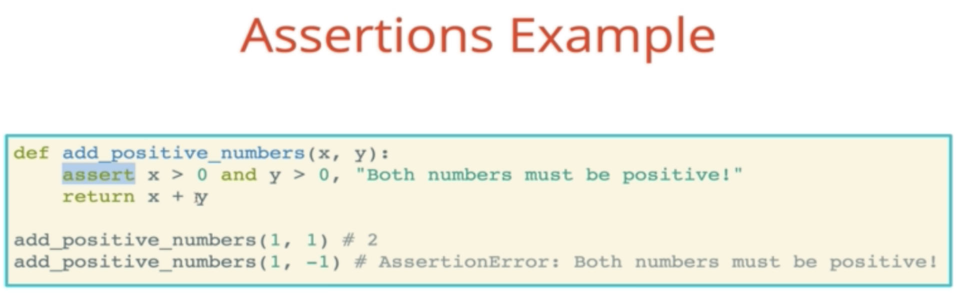
Test Driven Development—development begins by writing tests. Once tests are written, write code to make tests pass. Once tests pass, a feature is considered complete.

Red, Green, Refactor

1. Red—write a test that fails
2. Green—write the minimal amount of code necessary to make the test pass
3. Refactor—clean up the code, while ensuring that tests still pass

Assertions

- We can make simple assertions with the ‘assert’ keyword
- ‘assert’ accepts an expression
- Returns none if the expression is truthy
- Raises an AssertionError if the expression is falsey
- Accepts an optional error messages a second argument



Assertions Example

```python
def add_positive_numbers(x, y):
    assert x > 0 and y > 0, "Both numbers must be positive!"
    return x + y

add_positive_numbers(1, 1) # 2
add_positive_numbers(1, -1) # AssertionError: Both numbers must be positive!
```

Assertions Warning:

If a Python file is run with the -O flag, assertions will not be evaluated

doctests:

- We can write tests for functions inside of the docstring
- Write code that looks like it’s inside of a REPL

python3 -m doctest -v doctest_demo.py

Issues with doctests:

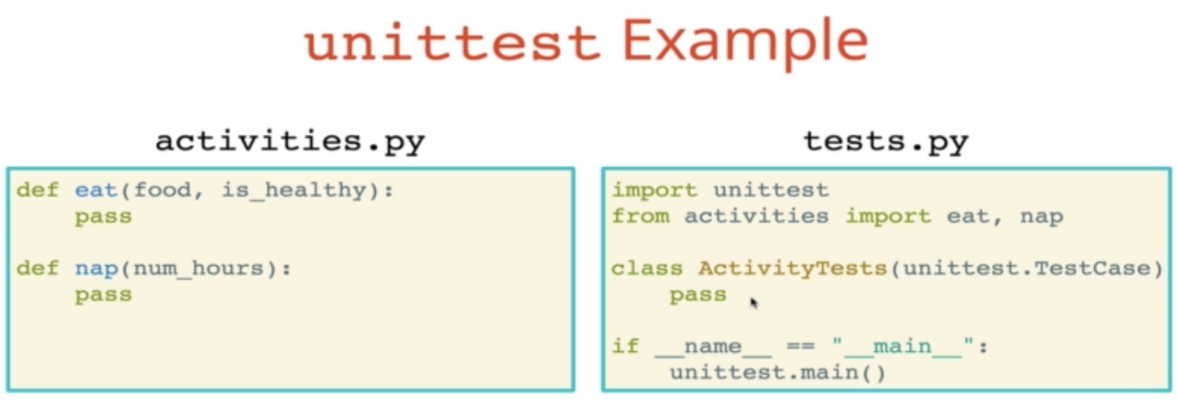
- Syntax is a little strange
- Clutters up our function code
- Lacks many features of larger testing tools

<u>Unit testing:</u>

- Test smallest parts of an application in isolation (i.e. units)
- Good candidates for unit testing: individual classes, modules, or functions
- Bad candidates for unit testing: an entire application, dependencies across several classes or modules

unittest—

- Python comes with a built-in module called unittest
- You can write unit tests encapsulated as classes that inherit from unittest.TestCase
- This inheritance gives you access to many assertion helpers that let you test the behavior of your functions
- You can run tests by calling unittest.main()



unittest Example

activities.py

```python
def eat(food, is_healthy):
    pass

def nap(num_hours):
    pass
```

tests.py

```python
import unittest
from activities import eat, nap

class ActivityTests(unittest.TestCase)
    pass

if __name__ == "__main__":
    unittest.main()
```

https://docs.python.org/3/library/unittest.html#test-cases

Method	Checks that	New in
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2

setUp and tearDown:

- For larger applications, you may want similar application state before running tests
- setUp runs before each test method
- tearDown runs after each test method
- Common use cases: adding/removing data from a test database, creating instances of a class

File I/O

Objectives:

- Read text files in PYthon
- Write text files in Python
- Use “with” blocks when reading/writing files
- Describe the different ways to open a file
- Read CSV files in Python
- Write CSV files in Python

Reading Files:

- You can read a file with the “open” function
- “open” returns a file object
- You can read a file object with the “read” method

Cursor Movement—Python reads files by using a cursor. This is like the cursor you see when you’re typing. After a file is read, the cursor is at the end.

- To move the cursor, we can use the “seek” method.

Closing a File:

- You can close a file with the close method
- You can check if a file is closed with the closed attribute
- Once closed, a file can’t be read again
- Always close files—it frees up system resources

When using the “with” keyword, we do not need to use close() to close the file because the following code will automatically close the file after:

```
with open(“example.txt”) as file:
        file.read()
```

Writing to Text Files:

- You can also use “open” to write to a file
- Need to specify the “w” flag as the second argument

```
        i.e. with open(“example.txt”, “w”) as file:
                file.write(“Writing in files is great.”)
```

Modes for Opening Files:

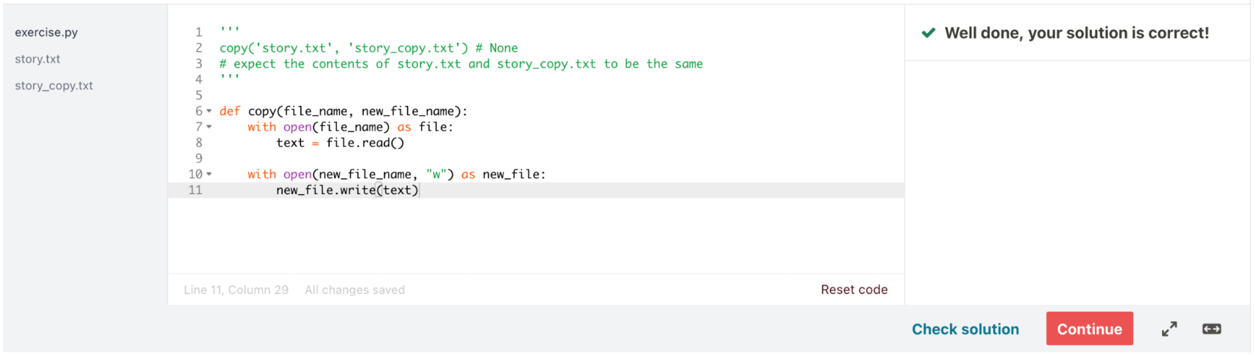
- r—Read a file (no writing). This is the default
- w—Write to a file (previous contents removed)
- a—Append to a file (previous contents not removed)
- r+—Read and write to a file (writing based on cursor)

copy

Write a function called **copy**, which takes in a file name and a new file name and copies the contents of the first file to the second file.

(Note: we've provided you with the first chapter of *Alice's Adventures in Wonderland* to give you some sample text to work with. This is also the text used in the tests.

```
copy('story.txt', 'story_copy.txt') # None
# expect the contents of story.txt and story_copy.txt to be the same
```



```python
'''
copy('story.txt', 'story_copy.txt') # None
# expect the contents of story.txt and story_copy.txt to be the same
'''

def copy(file_name, new_file_name):
    with open(file_name) as file:
        text = file.read()

    with open(new_file_name, "w") as new_file:
        new_file.write(text)
```

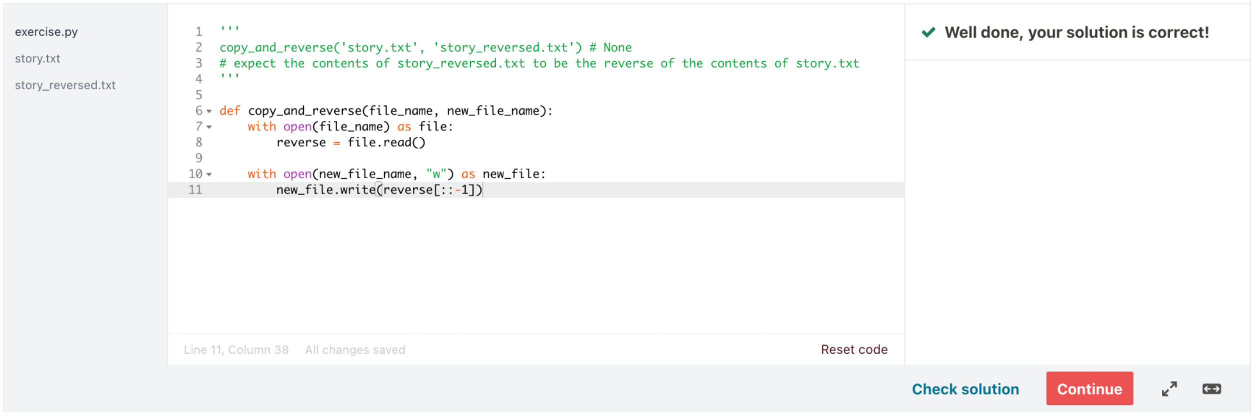
Line 11, Column 29 All changes saved

Well done, your solution is correct!

copy_and_reverse

Write a function called **copy_and_reverse**, which takes in a file name and a new file name and copies the **reversed** contents of the first file to the second file.

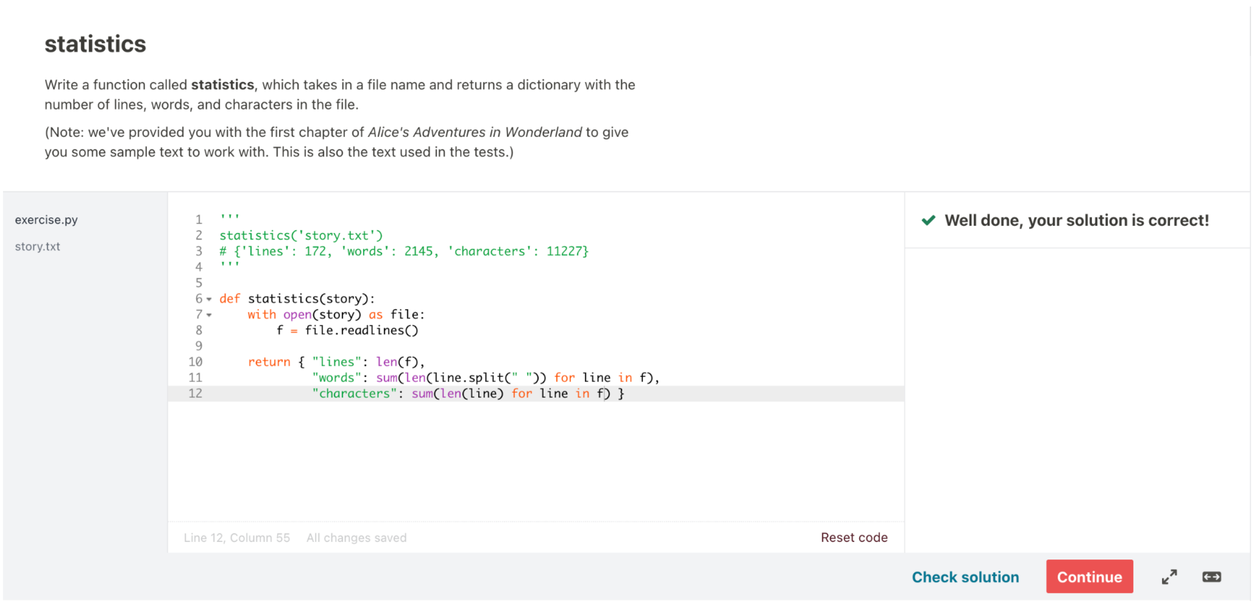
(Note: we've provided you with the first chapter of *Alice's Adventures in Wonderland* to give you some sample text to work with. This is also the text used in the tests.)



```python
'''
copy_and_reverse('story.txt', 'story_reversed.txt') # None
# expect the contents of story_reversed.txt to be the reverse of the contents of story.txt
'''

def copy_and_reverse(file_name, new_file_name):
    with open(file_name) as file:
        reverse = file.read()

    with open(new_file_name, "w") as new_file:
        new_file.write(reverse[::-1])
```

Line 11, Column 38 All changes saved

Well done, your solution is correct!



statistics

Write a function called **statistics**, which takes in a file name and returns a dictionary with the number of lines, words, and characters in the file.

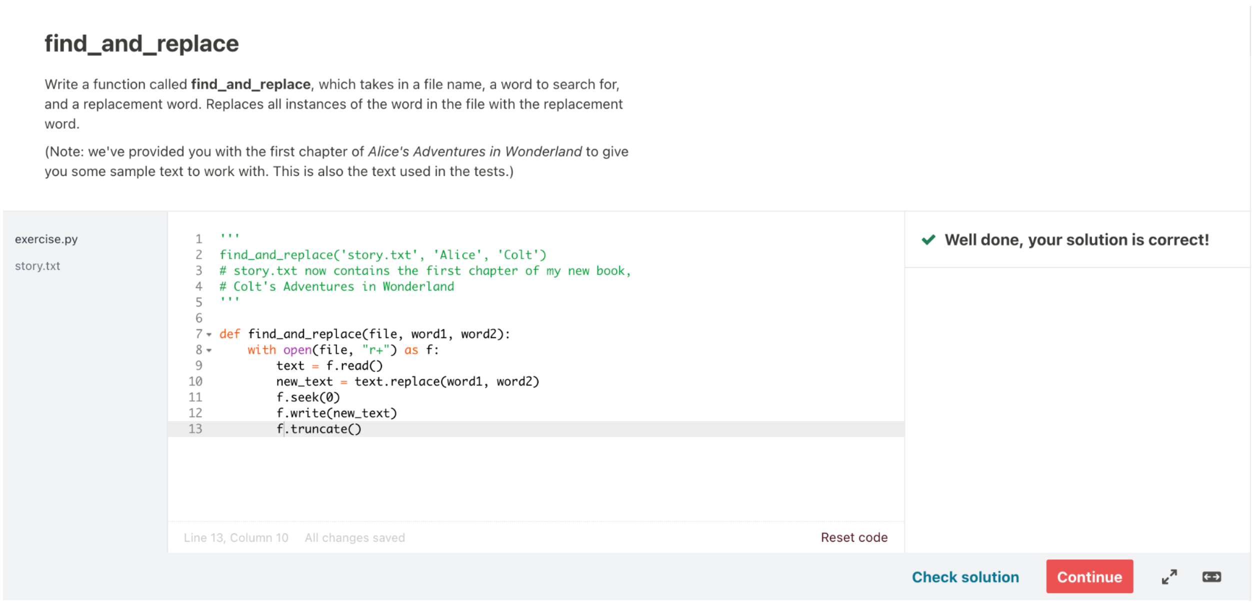
(Note: we've provided you with the first chapter of *Alice's Adventures in Wonderland* to give you some sample text to work with. This is also the text used in the tests.)

exercise.py
story.txt

```python
'''
statistics('story.txt')
# {'lines': 172, 'words': 2145, 'characters': 11227}
'''

def statistics(story):
    with open(story) as file:
        f = file.readlines()

    return { "lines": len(f),
             "words": sum(len(line.split(" ")) for line in f),
             "characters": sum(len(line) for line in f) }
```

Line 12, Column 55   All changes saved   Reset code

✓ Well done, your solution is correct!



find_and_replace

Write a function called **find_and_replace**, which takes in a file name, a word to search for, and a replacement word. Replaces all instances of the word in the file with the replacement word.

(Note: we've provided you with the first chapter of *Alice's Adventures in Wonderland* to give you some sample text to work with. This is also the text used in the tests.)

exercise.py
story.txt

```python
'''
find_and_replace('story.txt', 'Alice', 'Colt')
# story.txt now contains the first chapter of my new book,
# Colt's Adventures in Wonderland
'''

def find_and_replace(file, word1, word2):
    with open(file, "r+") as f:
        text = f.read()
        new_text = text.replace(word1, word2)
        f.seek(0)
        f.write(new_text)
        f.truncate()
```

Line 13, Column 10   All changes saved   Reset code

✓ Well done, your solution is correct!