Machine Learning Zoomcamp

Session #1.2

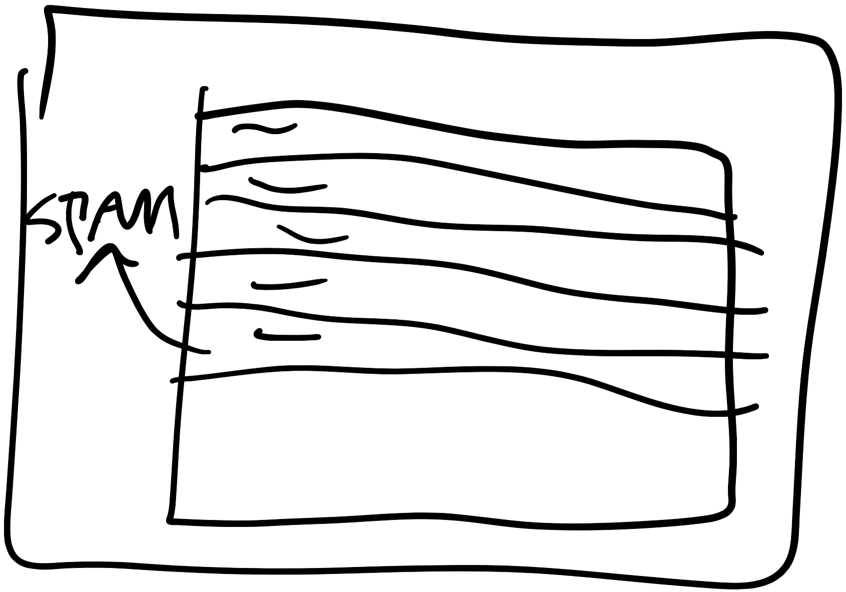
Machine Learning vs Rule-Based Systems

DataTalks.Club

Session #1.2: Plan

- A rule-based system for spam detection
- Using ML for spam detection
- Extracting features for ML

Email system



Spam

Subject: Get 50% off now
From: promotions@online.com

Whe
spa
you
click
Pay
with

Subject: URGENT: tax review
From: tax@online.com

Your tax review is pending acceptance. Review within 24 hours:

https://taxes.we-are-legit.com

Tax office.

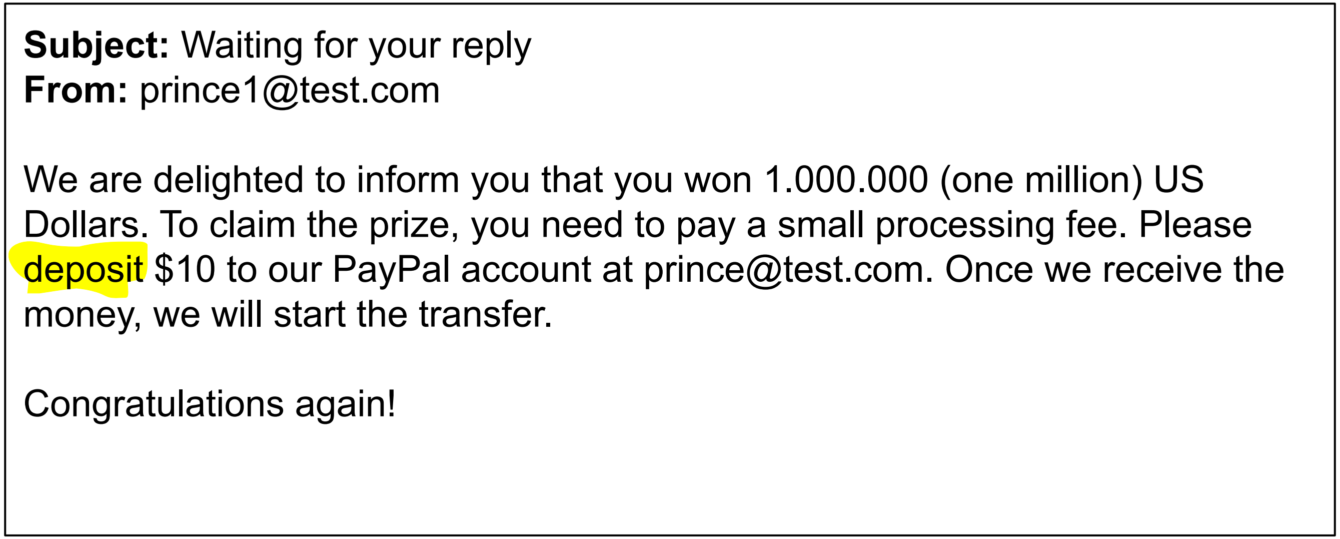
Rules

- If sender = promotions@online.com then “spam”
- If title contains “tax review” and sender domain is “online.com” then “spam”
- Otherwise, “good email”

Code

```
def detect_spam(email):
    if email.sender == 'promotions@online.com':
        return SPAM
    if contains(email.title, ['tax', 'rewiew']) and
            domain(email.sender, 'online.com'):
        return SPAM
    return GOOD
```

More



Subject: Waiting for your reply
From: prince1@test.com

We are delighted to inform you that you won 1.000.000 (one million) US Dollars. To claim the prize, you need to pay a small processing fee. Please deposit $10 to our PayPal account at prince@test.com. Once we receive the money, we will start the transfer.

Congratulations again!

Rules

- If sender = promotions@online.com then “spam”
- If title contains “tax review” and sender domain is “online.com” then “spam”
- **If body contains a word “deposit” then “spam”**
- Otherwise, “good email”

Code

```
def detect_spam(email):
    if email.sender == 'promotions@online.com':
        return SPAM
    if contains(email.title, ['tax', 'rewiew']) and
            domain(email.sender, 'online.com'):
        return SPAM
    if contains(email.body, ['deposit']):
        return SPAM
    return GOOD
```

More

Subject: Totally legit email
From: pedro@gmail.com

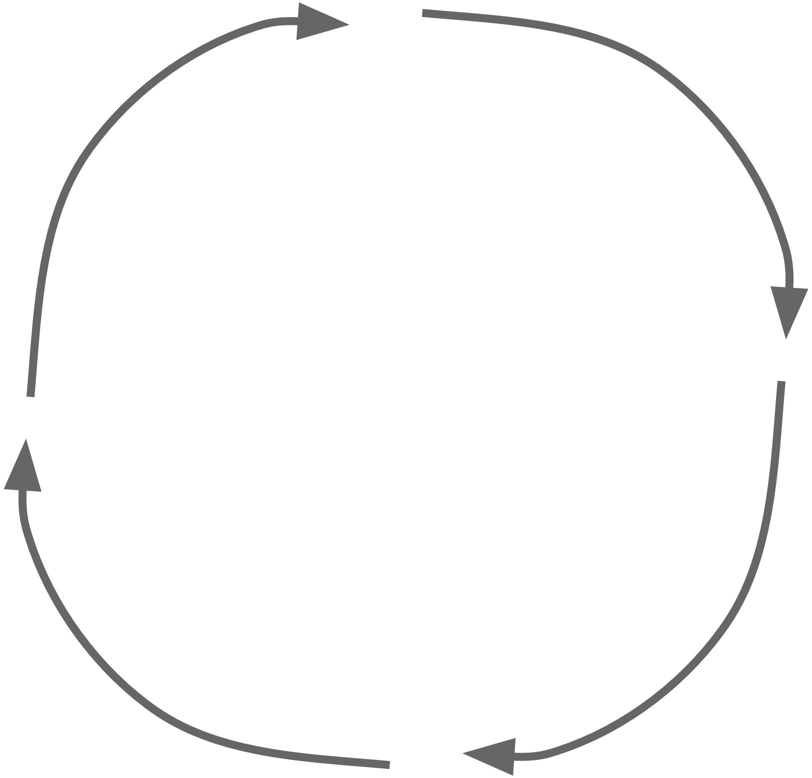
I transferred $50 to you one year ago, and now I’m moving out.
Please refund my deposit.

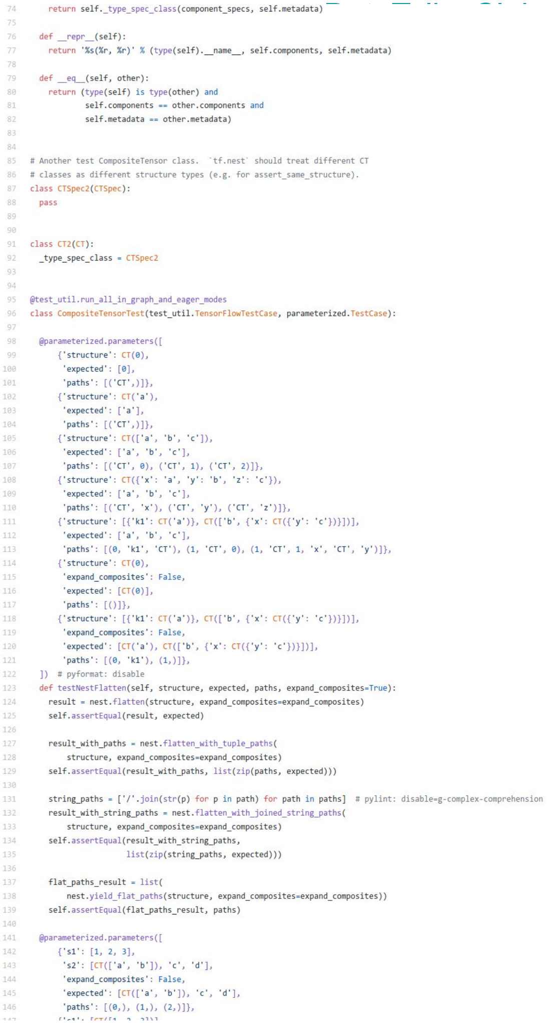
Pedro.

Rules

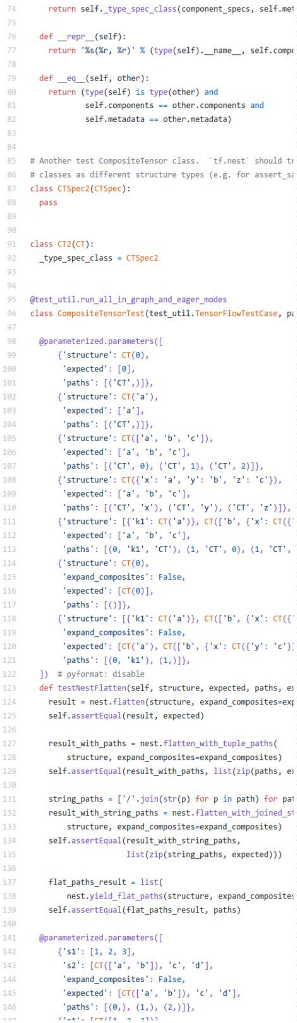
- If sender = promotions@online.com then “spam”
- If title contains “tax review” and sender domain is “online.com” then “spam”
- If body contains a word “deposit”
  - **If sender domain is “test.com” then “spam”**
  - **If body >= 100 words then spam**
- Otherwise, “good email”

Repeat

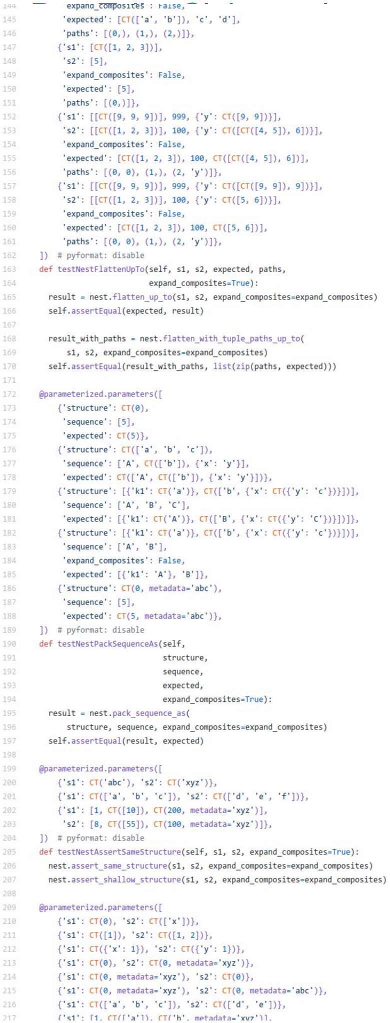




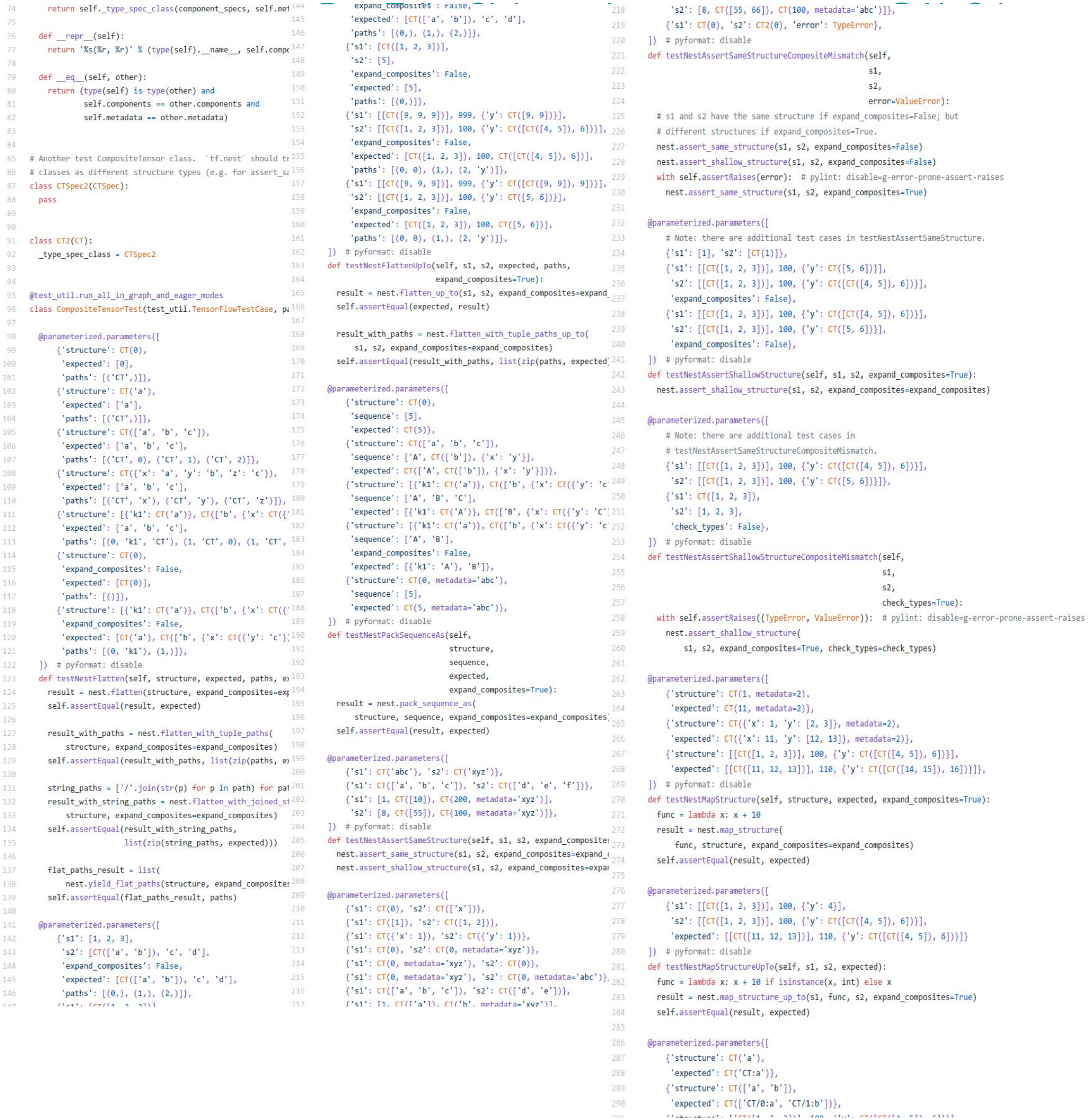
```python
    return self._type_spec_class(component_specs, self.metadata)

  def __repr__(self):
    return '%s(%r, %r)' % (type(self).__name__, self.components, self.metadata)

  def __eq__(self, other):
    return (type(self) is type(other) and
            self.components == other.components and
            self.metadata == other.metadata)


# Another test CompositeTensor class.  `tf.nest` should treat different CT
# classes as different structure types (e.g. for assert_same_structure).
class CTSpec2(CTSpec):
  pass


class CT2(CT):
  _type_spec_class = CTSpec2


@test_util.run_all_in_graph_and_eager_modes
class CompositeTensorTest(test_util.TensorFlowTestCase, parameterized.TestCase):

  @parameterized.parameters([
      {'structure': CT(0),
       'expected': [0],
       'paths': [('CT',)]},
      {'structure': CT('a'),
       'expected': ['a'],
       'paths': [('CT',)]},
      {'structure': CT(['a', 'b', 'c']),
       'expected': ['a', 'b', 'c'],
       'paths': [('CT', 0), ('CT', 1), ('CT', 2)]},
      {'structure': CT({'x': 'a', 'y': 'b', 'z': 'c'}),
       'expected': ['a', 'b', 'c'],
       'paths': [('CT', 'x'), ('CT', 'y'), ('CT', 'z')]},
      {'structure': [{'k1': CT('a')}, CT(['b', {'x': CT({'y': 'c'})}])],
       'expected': ['a', 'b', 'c'],
       'paths': [(0, 'k1', 'CT'), (1, 'CT', 0), (1, 'CT', 1, 'x', 'CT', 'y')]},
      {'structure': CT(0),
       'expand_composites': False,
       'expected': [CT(0)],
       'paths': [()]},
      {'structure': [{'k1': CT('a')}, CT(['b', {'x': CT({'y': 'c'})}])],
       'expand_composites': False,
       'expected': [CT('a'), CT(['b', {'x': CT({'y': 'c'})}])],
       'paths': [(0, 'k1'), (1,)]},
  ])  # pyformat: disable
  def testNestFlatten(self, structure, expected, paths, expand_composites=True):
    result = nest.flatten(structure, expand_composites=expand_composites)
    self.assertEqual(result, expected)

    result_with_paths = nest.flatten_with_tuple_paths(
        structure, expand_composites=expand_composites)
    self.assertEqual(result_with_paths, list(zip(paths, expected)))

    string_paths = ['/'.join(str(p) for p in path) for path in paths]  # pylint: disable=g-complex-comprehension
    result_with_string_paths = nest.flatten_with_joined_string_paths(
        structure, expand_composites=expand_composites)
    self.assertEqual(result_with_string_paths,
                     list(zip(string_paths, expected)))

    flat_paths_result = list(
        nest.yield_flat_paths(structure, expand_composites=expand_composites))
    self.assertEqual(flat_paths_result, paths)

  @parameterized.parameters([
      {'s1': [1, 2, 3],
       's2': [CT(['a', 'b']), 'c', 'd'],
       'expand_composites': False,
       'expected': [CT(['a', 'b']), 'c', 'd'],
       'paths': [(0,), (1,), (2,)]},
      {'s1': [CT([1, 2, 3])]
```



```
    return self._type_spec_class(component_specs, self.met

  def __repr__(self):
    return '%s(%r, %r)' % (type(self).__name__, self.compo

  def __eq__(self, other):
    return (type(self) is type(other) and
            self.components == other.components and
            self.metadata == other.metadata)


# Another test CompositeTensor class.  `tf.nest` should tr
# classes as different structure types (e.g. for assert_sa
class CTSpec2(CTSpec):
  pass


class CT2(CT):
  _type_spec_class = CTSpec2


@test_util.run_all_in_graph_and_eager_modes
class CompositeTensorTest(test_util.TensorFlowTestCase, pa

  @parameterized.parameters([
      {'structure': CT(0),
       'expected': [0],
       'paths': [('CT',)]},
      {'structure': CT('a'),
       'expected': ['a'],
       'paths': [('CT',)]},
      {'structure': CT(['a', 'b', 'c']),
       'expected': ['a', 'b', 'c'],
       'paths': [('CT', 0), ('CT', 1), ('CT', 2)]},
      {'structure': CT({'x': 'a', 'y': 'b', 'z': 'c'}),
       'expected': ['a', 'b', 'c'],
       'paths': [('CT', 'x'), ('CT', 'y'), ('CT', 'z')]},
      {'structure': [{'k1': CT('a')}, CT(['b', {'x': CT({'
       'expected': ['a', 'b', 'c'],
       'paths': [(0, 'k1', 'CT'), (1, 'CT', 0), (1, 'CT',
      {'structure': CT(0),
       'expand_composites': False,
       'expected': [CT(0)],
       'paths': [()]},
      {'structure': [{'k1': CT('a')}, CT(['b', {'x': CT({'
       'expand_composites': False,
       'expected': [CT('a'), CT(['b', {'x': CT({'y': 'c'})
       'paths': [(0, 'k1'), (1,)]},
  ])  # pyformat: disable
  def testNestFlatten(self, structure, expected, paths, ex
    result = nest.flatten(structure, expand_composites=exp
    self.assertEqual(result, expected)

    result_with_paths = nest.flatten_with_tuple_paths(
        structure, expand_composites=expand_composites)
    self.assertEqual(result_with_paths, list(zip(paths, ex

    string_paths = ['/'.join(str(p) for p in path) for pat
    result_with_string_paths = nest.flatten_with_joined_st
        structure, expand_composites=expand_composites)
    self.assertEqual(result_with_string_paths,
                     list(zip(string_paths, expected)))

    flat_paths_result = list(
        nest.yield_flat_paths(structure, expand_composites
    self.assertEqual(flat_paths_result, paths)

  @parameterized.parameters([
      {'s1': [1, 2, 3],
       's2': [CT(['a', 'b']), 'c', 'd'],
       'expand_composites': False,
       'expected': [CT(['a', 'b']), 'c', 'd'],
       'paths': [(0,), (1,), (2,)]},
      {'s1': [CT([1, 2, 3])]
```



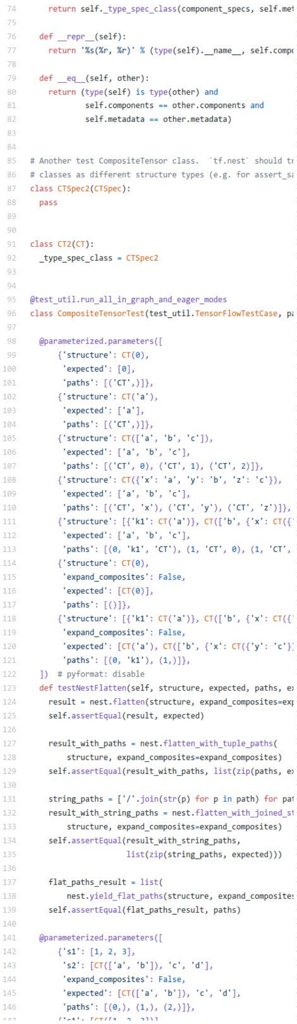
```
       'expand_composites': False,
       'expected': [CT(['a', 'b']), 'c', 'd'],
       'paths': [(0,), (1,), (2,)]},
      {'s1': [CT([1, 2, 3])],
       's2': [5],
       'expand_composites': False,
       'expected': [5],
       'paths': [(0,)]},
      {'s1': [[CT([9, 9, 9])], 999, {'y': CT([9, 9])}],
       's2': [[CT([1, 2, 3])], 100, {'y': CT([CT([4, 5]), 6])}],
       'expand_composites': False,
       'expected': [CT([1, 2, 3]), 100, CT([CT([4, 5]), 6])],
       'paths': [(0, 0), (1,), (2, 'y')]},
      {'s1': [[CT([9, 9, 9])], 999, {'y': CT([CT([9, 9]), 9])}],
       's2': [[CT([1, 2, 3])], 100, {'y': CT([5, 6])}],
       'expand_composites': False,
       'expected': [CT([1, 2, 3]), 100, CT([5, 6])],
       'paths': [(0, 0), (1,), (2, 'y')]},
  ])  # pyformat: disable
  def testNestFlattenUpTo(self, s1, s2, expected, paths,
                          expand_composites=True):
    result = nest.flatten_up_to(s1, s2, expand_composites=expand_composites)
    self.assertEqual(expected, result)

    result_with_paths = nest.flatten_with_tuple_paths_up_to(
        s1, s2, expand_composites=expand_composites)
    self.assertEqual(result_with_paths, list(zip(paths, expected)))

  @parameterized.parameters([
      {'structure': CT(0),
       'sequence': [5],
       'expected': CT(5)},
      {'structure': CT(['a', 'b', 'c']),
       'sequence': ['A', CT(['b']), {'x': 'y'}],
       'expected': CT(['A', CT(['b']), {'x': 'y'}])},
      {'structure': [{'k1': CT('a')}, CT(['b', {'x': CT({'y': 'c'})}])],
       'sequence': ['A', 'B', 'C'],
       'expected': [{'k1': CT('A')}, CT(['B', {'x': CT({'y': 'C'})}])]},
      {'structure': [{'k1': CT('a')}, CT(['b', {'x': CT({'y': 'c'})}])],
       'sequence': ['A', 'B'],
       'expand_composites': False,
       'expected': [{'k1': 'A'}, 'B']},
      {'structure': CT(0, metadata='abc'),
       'sequence': [5],
       'expected': CT(5, metadata='abc')},
  ])  # pyformat: disable
  def testNestPackSequenceAs(self,
                             structure,
                             sequence,
                             expected,
                             expand_composites=True):
    result = nest.pack_sequence_as(
        structure, sequence, expand_composites=expand_composites)
    self.assertEqual(result, expected)

  @parameterized.parameters([
      {'s1': CT('abc'), 's2': CT('xyz')},
      {'s1': CT(['a', 'b', 'c']), 's2': CT(['d', 'e', 'f'])},
      {'s1': [1, CT([10]), CT(200, metadata='xyz')],
       's2': [8, CT([55]), CT(100, metadata='xyz')]},
  ])  # pyformat: disable
  def testNestAssertSameStructure(self, s1, s2, expand_composites=True):
    nest.assert_same_structure(s1, s2, expand_composites=expand_composites)
    nest.assert_shallow_structure(s1, s2, expand_composites=expand_composites)

  @parameterized.parameters([
      {'s1': CT(0), 's2': CT(['x'])},
      {'s1': CT([1]), 's2': CT([1, 2])},
      {'s1': CT({'x': 1}), 's2': CT({'y': 1})},
      {'s1': CT(0), 's2': CT(0, metadata='xyz')},
      {'s1': CT(0, metadata='xyz'), 's2': CT(0)},
      {'s1': CT(0, metadata='xyz'), 's2': CT(0, metadata='abc')},
      {'s1': CT(['a', 'b', 'c']), 's2': CT(['d', 'e'])},
      {'s1': [1, CT(['a']), CT('b', metadata='xyz')],
```



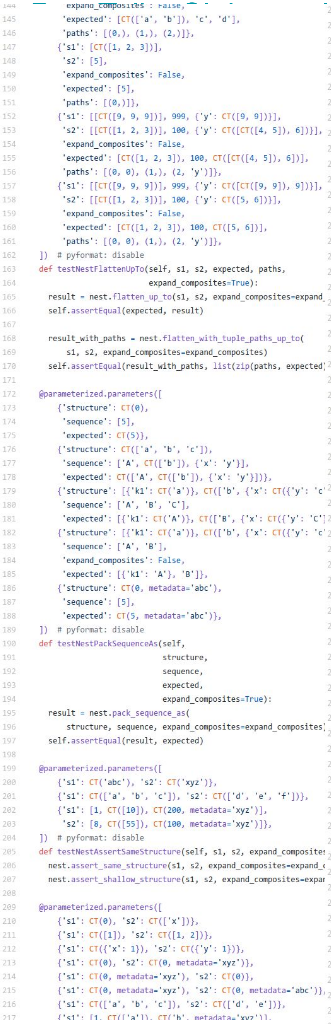
```
    return self._type_spec_class(component_specs, self.met

  def __repr__(self):
    return '%s(%r, %r)' % (type(self).__name__, self.compo

  def __eq__(self, other):
    return (type(self) is type(other) and
            self.components == other.components and
            self.metadata == other.metadata)


# Another test CompositeTensor class.  `tf.nest` should tr
# classes as different structure types (e.g. for assert_sa
class CTSpec2(CTSpec):
  pass


class CT2(CT):
  _type_spec_class = CTSpec2


@test_util.run_all_in_graph_and_eager_modes
class CompositeTensorTest(test_util.TensorFlowTestCase, pa

  @parameterized.parameters([
      {'structure': CT(0),
       'expected': [0],
       'paths': [('CT',)]},
      {'structure': CT('a'),
       'expected': ['a'],
       'paths': [('CT',)]},
      {'structure': CT(['a', 'b', 'c']),
       'expected': ['a', 'b', 'c'],
       'paths': [('CT', 0), ('CT', 1), ('CT', 2)]},
      {'structure': CT({'x': 'a', 'y': 'b', 'z': 'c'}),
       'expected': ['a', 'b', 'c'],
       'paths': [('CT', 'x'), ('CT', 'y'), ('CT', 'z')]},
      {'structure': [{'k1': CT('a')}, CT(['b', {'x': CT({'
       'expected': ['a', 'b', 'c'],
       'paths': [(0, 'k1', 'CT'), (1, 'CT', 0), (1, 'CT',
      {'structure': CT(0),
       'expand_composites': False,
       'expected': [CT(0)],
       'paths': [()]},
      {'structure': [{'k1': CT('a')}, CT(['b', {'x': CT({'
       'expand_composites': False,
       'expected': [CT('a'), CT(['b', {'x': CT({'y': 'c'}
       'paths': [(0, 'k1'), (1,)]},
  ])  # pyformat: disable
  def testNestFlatten(self, structure, expected, paths, ex
    result = nest.flatten(structure, expand_composites=exp
    self.assertEqual(result, expected)

    result_with_paths = nest.flatten_with_tuple_paths(
        structure, expand_composites=expand_composites)
    self.assertEqual(result_with_paths, list(zip(paths, ex

    string_paths = ['/'.join(str(p) for p in path) for pat
    result_with_string_paths = nest.flatten_with_joined_st
        structure, expand_composites=expand_composites)
    self.assertEqual(result_with_string_paths,
                     list(zip(string_paths, expected)))

    flat_paths_result = list(
        nest.yield_flat_paths(structure, expand_composites
    self.assertEqual(flat_paths_result, paths)

  @parameterized.parameters([
      {'s1': [1, 2, 3],
       's2': [CT(['a', 'b']), 'c', 'd'],
       'expand_composites': False,
       'expected': [CT(['a', 'b']), 'c', 'd'],
       'paths': [(0,), (1,), (2,)]},
```

```
       'expand_composites': False,
       'expected': [CT(['a', 'b']), 'c', 'd'],
       'paths': [(0,), (1,), (2,)]},
      {'s1': [CT([1, 2, 3])],
       's2': [5],
       'expand_composites': False,
       'expected': [5],
       'paths': [(0,)]},
      {'s1': [[CT([9, 9, 9])], 999, {'y': CT([9, 9])}],
       's2': [[CT([1, 2, 3])], 100, {'y': CT([CT([4, 5]), 6])}],
       'expand_composites': False,
       'expected': [CT([1, 2, 3]), 100, CT([CT([4, 5]), 6])],
       'paths': [(0, 0), (1,), (2, 'y')]},
      {'s1': [[CT([9, 9, 9])], 999, {'y': CT([CT([9, 9]), 9])}],
       's2': [[CT([1, 2, 3])], 100, {'y': CT([5, 6])}],
       'expand_composites': False,
       'expected': [CT([1, 2, 3]), 100, CT([5, 6])],
       'paths': [(0, 0), (1,), (2, 'y')]},
  ])  # pyformat: disable
  def testNestFlattenUpTo(self, s1, s2, expected, paths,
                          expand_composites=True):
    result = nest.flatten_up_to(s1, s2, expand_composites=expand_
    self.assertEqual(expected, result)

    result_with_paths = nest.flatten_with_tuple_paths_up_to(
        s1, s2, expand_composites=expand_composites)
    self.assertEqual(result_with_paths, list(zip(paths, expected)

  @parameterized.parameters([
      {'structure': CT(0),
       'sequence': [5],
       'expected': CT(5)},
      {'structure': CT(['a', 'b', 'c']),
       'sequence': ['A', CT(['b']), {'x': 'y'}],
       'expected': CT(['A', CT(['b']), {'x': 'y'}])},
      {'structure': [{'k1': CT('a')}, CT(['b', {'x': CT({'y': 'c'
       'sequence': ['A', 'B', 'C'],
       'expected': [{'k1': CT('A')}, CT(['B', {'x': CT({'y': 'C']
      {'structure': [{'k1': CT('a')}, CT(['b', {'x': CT({'y': 'c'
       'sequence': ['A', 'B'],
       'expand_composites': False,
       'expected': [{'k1': 'A'}, 'B']},
      {'structure': CT(0, metadata='abc'),
       'sequence': [5],
       'expected': CT(5, metadata='abc')},
  ])  # pyformat: disable
  def testNestPackSequenceAs(self,
                             structure,
                             sequence,
                             expected,
                             expand_composites=True):
    result = nest.pack_sequence_as(
        structure, sequence, expand_composites=expand_composites)
    self.assertEqual(result, expected)

  @parameterized.parameters([
      {'s1': CT('abc'), 's2': CT('xyz')},
      {'s1': CT(['a', 'b', 'c']), 's2': CT(['d', 'e', 'f'])},
      {'s1': [1, CT([10]), CT(200, metadata='xyz')],
       's2': [8, CT([55]), CT(100, metadata='xyz')]},
  ])  # pyformat: disable
  def testNestAssertSameStructure(self, s1, s2, expand_composites
    nest.assert_same_structure(s1, s2, expand_composites=expand_c
    nest.assert_shallow_structure(s1, s2, expand_composites=expar

  @parameterized.parameters([
      {'s1': CT(0), 's2': CT(['x'])},
      {'s1': CT([1]), 's2': CT([1, 2])},
      {'s1': CT({'x': 1}), 's2': CT({'y': 1})},
      {'s1': CT(0), 's2': CT(0, metadata='xyz')},
      {'s1': CT(0, metadata='xyz'), 's2': CT(0)},
      {'s1': CT(0, metadata='xyz'), 's2': CT(0, metadata='abc')},
      {'s1': CT(['a', 'b', 'c']), 's2': CT(['d', 'e'])},
      {'s1': [1, CT(['a']), CT('b', metadata='xyz')],
```

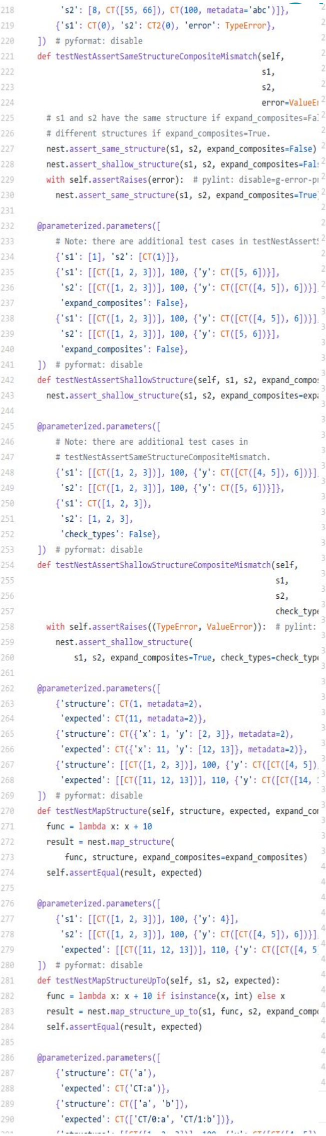
```
       's2': [8, CT([55, 66]), CT(100, metadata='abc')]},
      {'s1': CT(0), 's2': CT2(0), 'error': TypeError},
  ])  # pyformat: disable
  def testNestAssertSameStructureCompositeMismatch(self,
                                                   s1,
                                                   s2,
                                                   error=ValueError):
    # s1 and s2 have the same structure if expand_composites=False; but
    # different structures if expand_composites=True.
    nest.assert_same_structure(s1, s2, expand_composites=False)
    nest.assert_shallow_structure(s1, s2, expand_composites=False)
    with self.assertRaises(error):  # pylint: disable=g-error-prone-assert-raises
      nest.assert_same_structure(s1, s2, expand_composites=True)

  @parameterized.parameters([
      # Note: there are additional test cases in testNestAssertSameStructure.
      {'s1': [1], 's2': [CT(1)]},
      {'s1': [[CT([1, 2, 3])], 100, {'y': CT([5, 6])}],
       's2': [[CT([1, 2, 3])], 100, {'y': CT([CT([4, 5]), 6])}],
       'expand_composites': False},
      {'s1': [[CT([1, 2, 3])], 100, {'y': CT([CT([4, 5]), 6])}],
       's2': [[CT([1, 2, 3])], 100, {'y': CT([5, 6])}],
       'expand_composites': False},
  ])  # pyformat: disable
  def testNestAssertShallowStructure(self, s1, s2, expand_composites=True):
    nest.assert_shallow_structure(s1, s2, expand_composites=expand_composites)

  @parameterized.parameters([
      # Note: there are additional test cases in
      # testNestAssertSameStructureCompositeMismatch.
      {'s1': [[CT([1, 2, 3])], 100, {'y': CT([CT([4, 5]), 6])}],
       's2': [[CT([1, 2, 3])], 100, {'y': CT([5, 6])}]},
      {'s1': CT([1, 2, 3]),
       's2': [1, 2, 3],
       'check_types': False},
  ])  # pyformat: disable
  def testNestAssertShallowStructureCompositeMismatch(self,
                                                      s1,
                                                      s2,
                                                      check_types=True):
    with self.assertRaises((TypeError, ValueError)):  # pylint: disable=g-error-prone-assert-raises
      nest.assert_shallow_structure(
          s1, s2, expand_composites=True, check_types=check_types)

  @parameterized.parameters([
      {'structure': CT(1, metadata=2),
       'expected': CT(11, metadata=2)},
      {'structure': CT({'x': 1, 'y': [2, 3]}, metadata=2),
       'expected': CT({'x': 11, 'y': [12, 13]}, metadata=2)},
      {'structure': [[CT([1, 2, 3])], 100, {'y': CT([CT([4, 5]), 6])}],
       'expected': [[CT([11, 12, 13])], 110, {'y': CT([CT([14, 15]), 16])}]},
  ])  # pyformat: disable
  def testNestMapStructure(self, structure, expected, expand_composites=True):
    func = lambda x: x + 10
    result = nest.map_structure(
        func, structure, expand_composites=expand_composites)
    self.assertEqual(result, expected)

  @parameterized.parameters([
      {'s1': [[CT([1, 2, 3])], 100, {'y': 4}],
       's2': [[CT([1, 2, 3])], 100, {'y': CT([CT([4, 5]), 6])}],
       'expected': [[CT([11, 12, 13])], 110, {'y': CT([CT([4, 5]), 6])}]}
  ])  # pyformat: disable
  def testNestMapStructureUpTo(self, s1, s2, expected):
    func = lambda x: x + 10 if isinstance(x, int) else x
    result = nest.map_structure_up_to(s1, func, s2, expand_composites=True)
    self.assertEqual(result, expected)

  @parameterized.parameters([
      {'structure': CT('a'),
       'expected': CT('CT:a')},
      {'structure': CT(['a', 'b']),
       'expected': CT(['CT/0:a', 'CT/1:b'])},
```



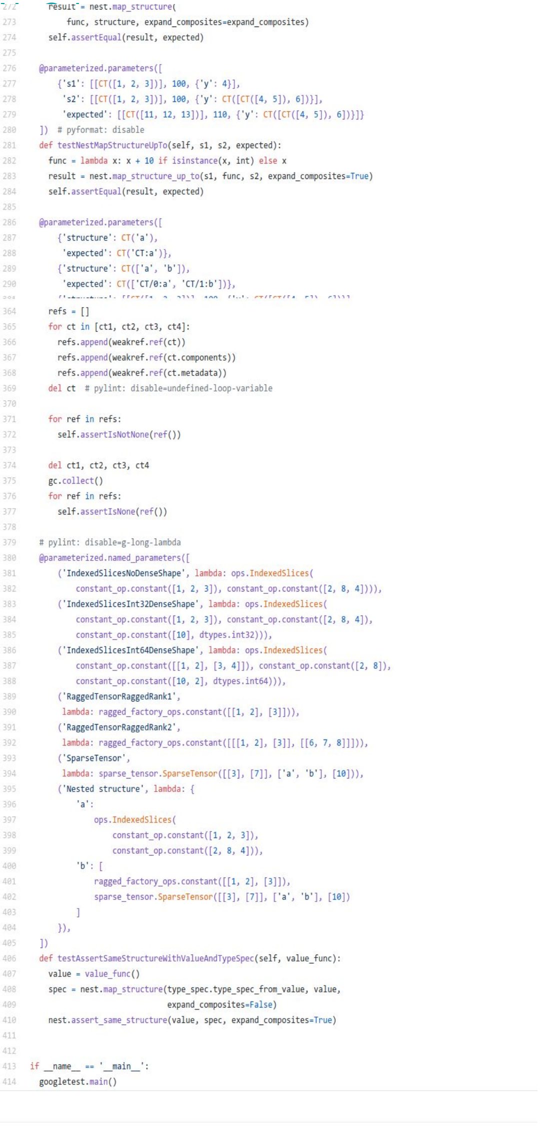
```
    return self._type_spec_class(component_specs, self.met

  def __repr__(self):
    return '%s(%r, %r)' % (type(self).__name__, self.compo

  def __eq__(self, other):
    return (type(self) is type(other) and
            self.components == other.components and
            self.metadata == other.metadata)


# Another test CompositeTensor class.  `tf.nest` should tr
# classes as different structure types (e.g. for assert_sa
class CTSpec2(CTSpec):
  pass


class CT2(CT):
  _type_spec_class = CTSpec2


@test_util.run_all_in_graph_and_eager_modes
class CompositeTensorTest(test_util.TensorFlowTestCase, pa

  @parameterized.parameters([
      {'structure': CT(0),
       'expected': [0],
       'paths': [('CT',)]},
      {'structure': CT('a'),
       'expected': ['a'],
       'paths': [('CT',)]},
      {'structure': CT(['a', 'b', 'c']),
       'expected': ['a', 'b', 'c'],
       'paths': [('CT', 0), ('CT', 1), ('CT', 2)]},
      {'structure': CT({'x': 'a', 'y': 'b', 'z': 'c'}),
       'expected': ['a', 'b', 'c'],
       'paths': [('CT', 'x'), ('CT', 'y'), ('CT', 'z')]},
      {'structure': [{'k1': CT('a')}, CT(['b', {'x': CT({'
       'expected': ['a', 'b', 'c'],
       'paths': [(0, 'k1', 'CT'), (1, 'CT', 0), (1, 'CT',
      {'structure': CT(0),
       'expand_composites': False,
       'expected': [CT(0)],
       'paths': [()]},
      {'structure': [{'k1': CT('a')}, CT(['b', {'x': CT({'
       'expand_composites': False,
       'expected': [CT('a'), CT(['b', {'x': CT({'y': 'c'})
       'paths': [(0, 'k1'), (1,)]},
  ])  # pyformat: disable
  def testNestFlatten(self, structure, expected, paths, ex
    result = nest.flatten(structure, expand_composites=exp
    self.assertEqual(result, expected)

    result_with_paths = nest.flatten_with_tuple_paths(
        structure, expand_composites=expand_composites)
    self.assertEqual(result_with_paths, list(zip(paths, ex

    string_paths = ['/'.join(str(p) for p in path) for pat
    result_with_string_paths = nest.flatten_with_joined_st
        structure, expand_composites=expand_composites)
    self.assertEqual(result_with_string_paths,
                     list(zip(string_paths, expected)))

    flat_paths_result = list(
        nest.yield_flat_paths(structure, expand_composite
    self.assertEqual(flat_paths_result, paths)

  @parameterized.parameters([
      {'s1': [1, 2, 3],
       's2': [CT(['a', 'b']), 'c', 'd'],
       'expand_composites': False,
       'expected': [CT(['a', 'b']), 'c', 'd'],
       'paths': [(0,), (1,), (2,)]},
      {'s1': [CT([1, 2, 3])],
```



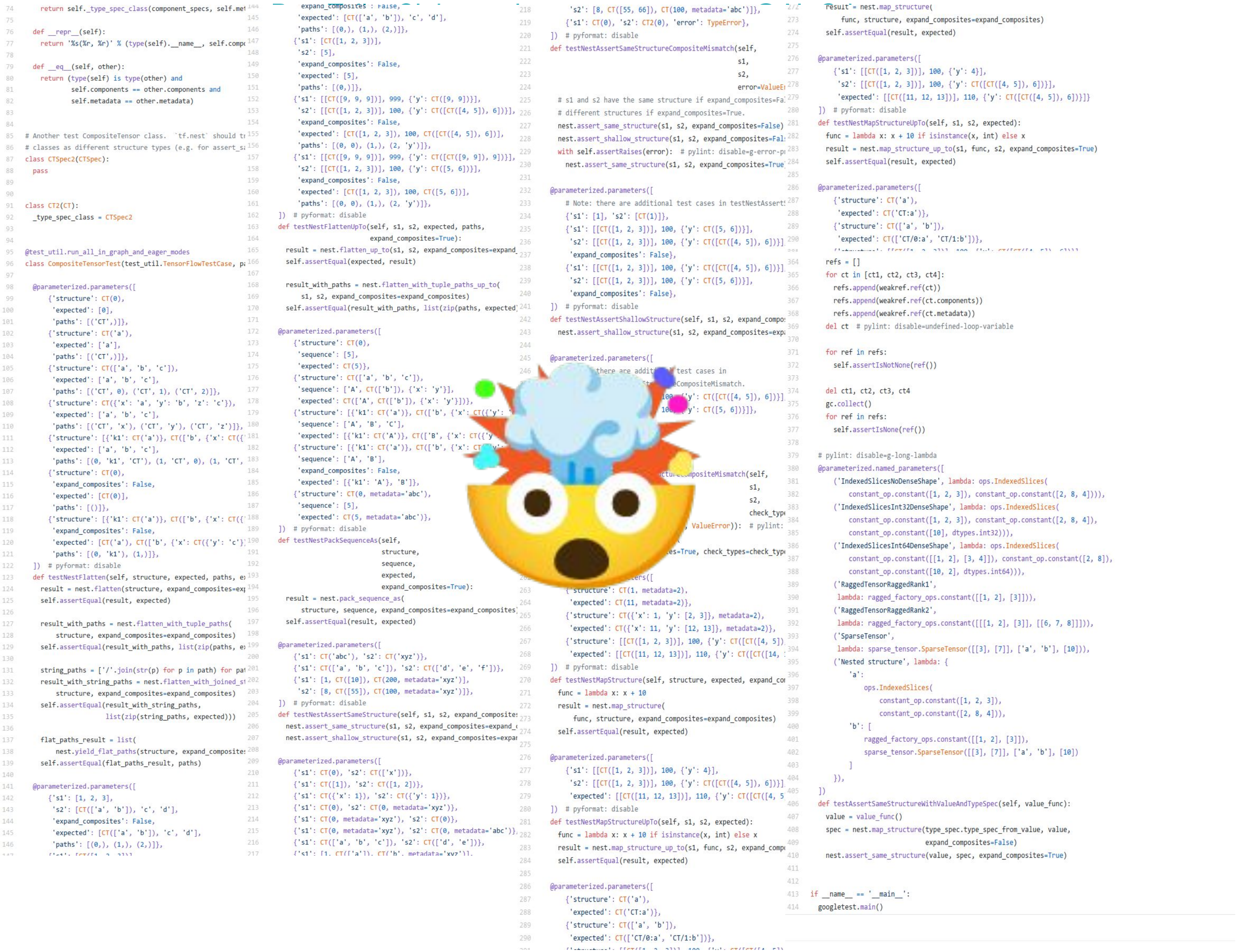
```
       'expand_composites': False,
       'expected': [CT(['a', 'b']), 'c', 'd'],
       'paths': [(0,), (1,), (2,)]},
      {'s1': [CT([1, 2, 3])],
       's2': [5],
       'expand_composites': False,
       'expected': [5],
       'paths': [(0,)]},
      {'s1': [[CT([9, 9, 9])], 999, {'y': CT([9, 9])}],
       's2': [[CT([1, 2, 3])], 100, {'y': CT([CT([4, 5]), 6])}],
       'expand_composites': False,
       'expected': [CT([1, 2, 3]), 100, CT([CT([4, 5]), 6])],
       'paths': [(0, 0), (1,), (2, 'y')]},
      {'s1': [[CT([9, 9, 9])], 999, {'y': CT([CT([9, 9]), 9])}],
       's2': [[CT([1, 2, 3])], 100, {'y': CT([5, 6])}],
       'expand_composites': False,
       'expected': [CT([1, 2, 3]), 100, CT([5, 6])],
       'paths': [(0, 0), (1,), (2, 'y')]},
  ])  # pyformat: disable
  def testNestFlattenUpTo(self, s1, s2, expected, paths,
                          expand_composites=True):
    result = nest.flatten_up_to(s1, s2, expand_composites=expand_
    self.assertEqual(expected, result)

    result_with_paths = nest.flatten_with_tuple_paths_up_to(
        s1, s2, expand_composites=expand_composites)
    self.assertEqual(result_with_paths, list(zip(paths, expected)

  @parameterized.parameters([
      {'structure': CT(0),
       'sequence': [5],
       'expected': CT(5)},
      {'structure': CT(['a', 'b', 'c']),
       'sequence': ['A', CT(['b']), {'x': 'y'}],
       'expected': CT(['A', CT(['b']), {'x': 'y'}])},
      {'structure': [{'k1': CT('a')}, CT(['b', {'x': CT({'y': 'c
       'sequence': ['A', 'B', 'C'],
       'expected': [{'k1': CT('A')}, CT(['B', {'x': CT({'y': 'C']
      {'structure': [{'k1': CT('a')}, CT(['b', {'x': CT({'y': 'c
       'sequence': ['A', 'B'],
       'expand_composites': False,
       'expected': [{'k1': 'A'}, 'B']},
      {'structure': CT(0, metadata='abc'),
       'sequence': [5],
       'expected': CT(5, metadata='abc')},
  ])  # pyformat: disable
  def testNestPackSequenceAs(self,
                             structure,
                             sequence,
                             expected,
                             expand_composites=True):
    result = nest.pack_sequence_as(
        structure, sequence, expand_composites=expand_composites)
    self.assertEqual(result, expected)

  @parameterized.parameters([
      {'s1': CT('abc'), 's2': CT('xyz')},
      {'s1': CT(['a', 'b', 'c']), 's2': CT(['d', 'e', 'f'])},
      {'s1': [1, CT([10]), CT(200, metadata='xyz')],
       's2': [8, CT([55]), CT(100, metadata='xyz')]},
  ])  # pyformat: disable
  def testNestAssertSameStructure(self, s1, s2, expand_composites
    nest.assert_same_structure(s1, s2, expand_composites=expand_c
    nest.assert_shallow_structure(s1, s2, expand_composites=expar

  @parameterized.parameters([
      {'s1': CT(0), 's2': CT(['x'])},
      {'s1': CT([1]), 's2': CT([1, 2])},
      {'s1': CT({'x': 1}), 's2': CT({'y': 1})},
      {'s1': CT(0), 's2': CT(0, metadata='xyz')},
      {'s1': CT(0, metadata='xyz'), 's2': CT(0)},
      {'s1': CT(0, metadata='xyz'), 's2': CT(0, metadata='abc')},
      {'s1': CT(['a', 'b', 'c']), 's2': CT(['d', 'e'])},
      {'s1': [1, CT(['a']), CT('b', metadata='xyz')],
```



```
       's2': [8, CT([55, 66]), CT(100, metadata='abc')]},
      {'s1': CT(0), 's2': CT2(0), 'error': TypeError},
  ])  # pyformat: disable
  def testNestAssertSameStructureCompositeMismatch(self,
                                                   s1,
                                                   s2,
                                                   error=ValueEr
    # s1 and s2 have the same structure if expand_composites=Fal
    # different structures if expand_composites=True.
    nest.assert_same_structure(s1, s2, expand_composites=False)
    nest.assert_shallow_structure(s1, s2, expand_composites=Fals
    with self.assertRaises(error):  # pylint: disable=g-error-pr
      nest.assert_same_structure(s1, s2, expand_composites=True)

  @parameterized.parameters([
      # Note: there are additional test cases in testNestAssertS
      {'s1': [1], 's2': [CT(1)]},
      {'s1': [[CT([1, 2, 3])], 100, {'y': CT([5, 6])}],
       's2': [[CT([1, 2, 3])], 100, {'y': CT([CT([4, 5]), 6])}].
       'expand_composites': False},
      {'s1': [[CT([1, 2, 3])], 100, {'y': CT([CT([4, 5]), 6])}].
       's2': [[CT([1, 2, 3])], 100, {'y': CT([5, 6])}],
       'expand_composites': False},
  ])  # pyformat: disable
  def testNestAssertShallowStructure(self, s1, s2, expand_compos
    nest.assert_shallow_structure(s1, s2, expand_composites=expa

  @parameterized.parameters([
      # Note: there are additional test cases in
      # testNestAssertSameStructureCompositeMismatch.
      {'s1': [[CT([1, 2, 3])], 100, {'y': CT([CT([4, 5]), 6])}].
       's2': [[CT([1, 2, 3])], 100, {'y': CT([5, 6])}]},
      {'s1': CT([1, 2, 3]),
       's2': [1, 2, 3],
       'check_types': False},
  ])  # pyformat: disable
  def testNestAssertShallowStructureCompositeMismatch(self,
                                                      s1,
                                                      s2,
                                                      check_type

    with self.assertRaises((TypeError, ValueError)):  # pylint:
      nest.assert_shallow_structure(
          s1, s2, expand_composites=True, check_types=check_type

  @parameterized.parameters([
      {'structure': CT(1, metadata=2),
       'expected': CT(11, metadata=2)},
      {'structure': CT({'x': 1, 'y': [2, 3]}, metadata=2),
       'expected': CT({'x': 11, 'y': [12, 13]}, metadata=2)},
      {'structure': [[CT([1, 2, 3])], 100, {'y': CT([CT([4, 5]).
       'expected': [[CT([11, 12, 13])], 110, {'y': CT([CT([14, 
  ])  # pyformat: disable
  def testNestMapStructure(self, structure, expected, expand_co
    func = lambda x: x + 10
    result = nest.map_structure(
        func, structure, expand_composites=expand_composites)
    self.assertEqual(result, expected)

  @parameterized.parameters([
      {'s1': [[CT([1, 2, 3])], 100, {'y': 4}],
       's2': [[CT([1, 2, 3])], 100, {'y': CT([CT([4, 5]), 6])}].
       'expected': [[CT([11, 12, 13])], 110, {'y': CT([CT([4, 5
  ])  # pyformat: disable
  def testNestMapStructureUpTo(self, s1, s2, expected):
    func = lambda x: x + 10 if isinstance(x, int) else x
    result = nest.map_structure_up_to(s1, func, s2, expand_compo
    self.assertEqual(result, expected)

  @parameterized.parameters([
      {'structure': CT('a'),
       'expected': CT('CT:a')},
      {'structure': CT(['a', 'b']),
       'expected': CT(['CT/0:a', 'CT/1:b'])},
```



```
    result = nest.map_structure(
        func, structure, expand_composites=expand_composites)
    self.assertEqual(result, expected)

  @parameterized.parameters([
      {'s1': [[CT([1, 2, 3])], 100, {'y': 4}],
       's2': [[CT([1, 2, 3])], 100, {'y': CT([CT([4, 5]), 6])}],
       'expected': [[CT([11, 12, 13])], 110, {'y': CT([CT([4, 5]), 6])}]}
  ])  # pyformat: disable
  def testNestMapStructureUpTo(self, s1, s2, expected):
    func = lambda x: x + 10 if isinstance(x, int) else x
    result = nest.map_structure_up_to(s1, func, s2, expand_composites=True)
    self.assertEqual(result, expected)

  @parameterized.parameters([
      {'structure': CT('a'),
       'expected': CT('CT:a')},
      {'structure': CT(['a', 'b']),
       'expected': CT(['CT/0:a', 'CT/1:b'])},
    refs = []
    for ct in [ct1, ct2, ct3, ct4]:
      refs.append(weakref.ref(ct))
      refs.append(weakref.ref(ct.components))
      refs.append(weakref.ref(ct.metadata))
    del ct  # pylint: disable=undefined-loop-variable

    for ref in refs:
      self.assertIsNotNone(ref())

    del ct1, ct2, ct3, ct4
    gc.collect()
    for ref in refs:
      self.assertIsNone(ref())

  # pylint: disable=g-long-lambda
  @parameterized.named_parameters([
      ('IndexedSlicesNoDenseShape', lambda: ops.IndexedSlices(
          constant_op.constant([1, 2, 3]), constant_op.constant([2, 8, 4]))),
      ('IndexedSlicesInt32DenseShape', lambda: ops.IndexedSlices(
          constant_op.constant([1, 2, 3]), constant_op.constant([2, 8, 4]),
          constant_op.constant([10], dtypes.int32))),
      ('IndexedSlicesInt64DenseShape', lambda: ops.IndexedSlices(
          constant_op.constant([[1, 2], [3, 4]]), constant_op.constant([2, 8]),
          constant_op.constant([10, 2], dtypes.int64))),
      ('RaggedTensorRaggedRank1',
       lambda: ragged_factory_ops.constant([[1, 2], [3]])),
      ('RaggedTensorRaggedRank2',
       lambda: ragged_factory_ops.constant([[[1, 2], [3]], [[6, 7, 8]]])),
      ('SparseTensor',
       lambda: sparse_tensor.SparseTensor([[3], [7]], ['a', 'b'], [10])),
      ('Nested structure', lambda: {
          'a':
              ops.IndexedSlices(
                  constant_op.constant([1, 2, 3]),
                  constant_op.constant([2, 8, 4])),
          'b': [
              ragged_factory_ops.constant([[1, 2], [3]]),
              sparse_tensor.SparseTensor([[3], [7]], ['a', 'b'], [10])
          ]
      }),
  ])
  def testAssertSameStructureWithValueAndTypeSpec(self, value_func):
    value = value_func()
    spec = nest.map_structure(type_spec.type_spec_from_value, value,
                              expand_composites=False)
    nest.assert_same_structure(value, spec, expand_composites=True)


if __name__ == '__main__':
  googletest.main()
```

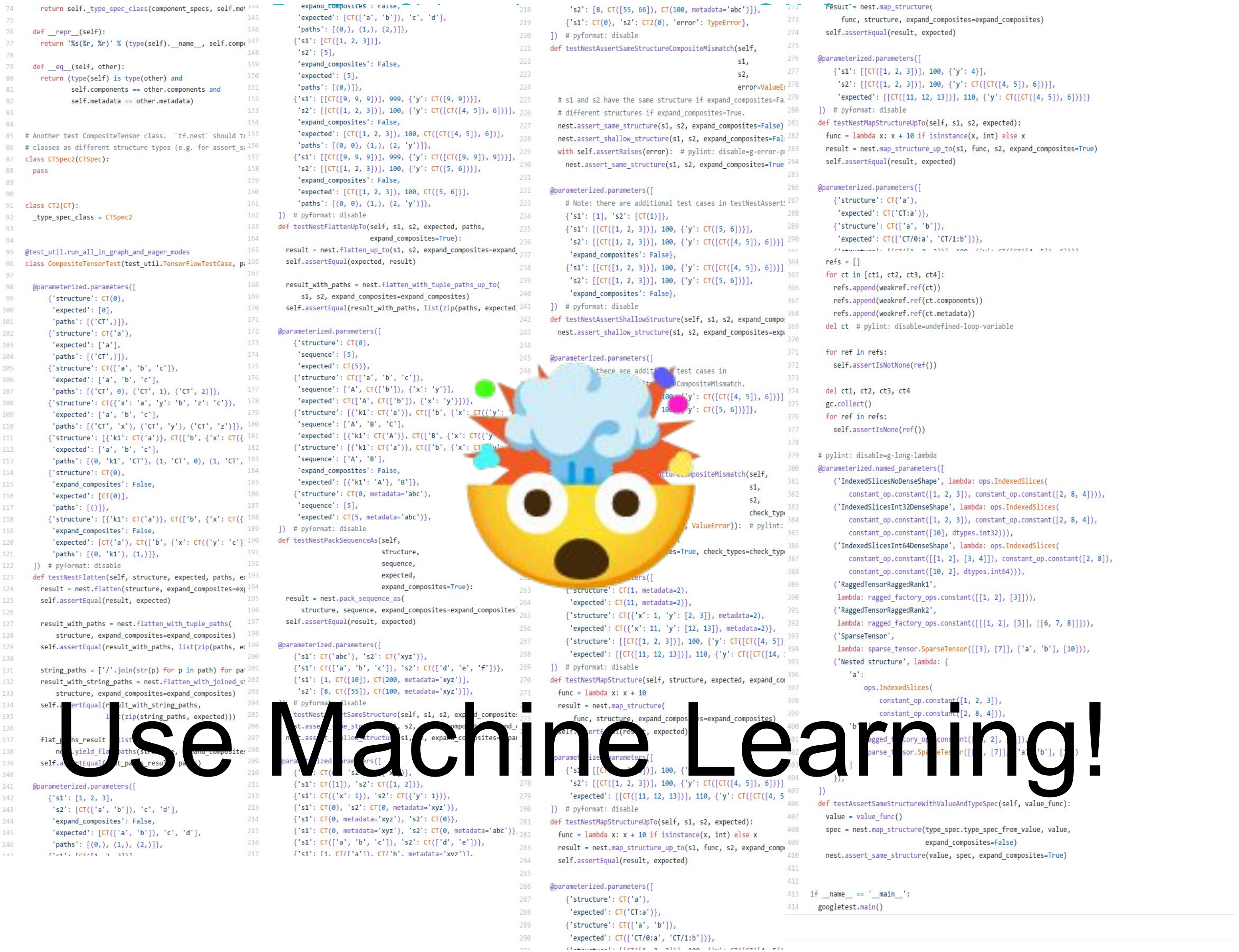


```python
    return self._type_spec_class(component_specs, self.met

  def __repr__(self):
    return '%s(%r, %r)' % (type(self).__name__, self.compo

  def __eq__(self, other):
    return (type(self) is type(other) and
            self.components == other.components and
            self.metadata == other.metadata)


# Another test CompositeTensor class.  `tf.nest` should tr
# classes as different structure types (e.g. for assert_sa
class CTSpec2(CTSpec):
  pass


class CT2(CT):
  _type_spec_class = CTSpec2


@test_util.run_all_in_graph_and_eager_modes
class CompositeTensorTest(test_util.TensorFlowTestCase, pa

  @parameterized.parameters([
      {'structure': CT(0),
       'expected': [0],
       'paths': [('CT',)]},
      {'structure': CT('a'),
       'expected': ['a'],
       'paths': [('CT',)]},
      {'structure': CT(['a', 'b', 'c']),
       'expected': ['a', 'b', 'c'],
       'paths': [('CT', 0), ('CT', 1), ('CT', 2)]},
      {'structure': CT({'x': 'a', 'y': 'b', 'z': 'c'}),
       'expected': ['a', 'b', 'c'],
       'paths': [('CT', 'x'), ('CT', 'y'), ('CT', 'z')]},
      {'structure': [{'k1': CT('a')}, CT(['b', {'x': CT({'
       'expected': ['a', 'b', 'c'],
       'paths': [(0, 'k1', 'CT'), (1, 'CT', 0), (1, 'CT',
      {'structure': CT(0),
       'expand_composites': False,
       'expected': [CT(0)],
       'paths': [()]},
      {'structure': [{'k1': CT('a')}, CT(['b', {'x': CT({'
       'expand_composites': False,
       'expected': [CT('a'), CT(['b', {'x': CT({'y': 'c'})
       'paths': [(0, 'k1'), (1,)]},
  ])  # pyformat: disable
  def testNestFlatten(self, structure, expected, paths, ex
    result = nest.flatten(structure, expand_composites=exp
    self.assertEqual(result, expected)

    result_with_paths = nest.flatten_with_tuple_paths(
        structure, expand_composites=expand_composites)
    self.assertEqual(result_with_paths, list(zip(paths, ex

    string_paths = ['/'.join(str(p) for p in path) for pat
    result_with_string_paths = nest.flatten_with_joined_st
        structure, expand_composites=expand_composites)
    self.assertEqual(result_with_string_paths,
                     list(zip(string_paths, expected)))

    flat_paths_result = list(
        nest.yield_flat_paths(structure, expand_composites
    self.assertEqual(flat_paths_result, paths)

  @parameterized.parameters([
      {'s1': [1, 2, 3],
       's2': [CT(['a', 'b']), 'c', 'd'],
       'expand_composites': False,
       'expected': [CT(['a', 'b']), 'c', 'd'],
       'paths': [(0,), (1,), (2,)]},
```

```python
       'expand_composites': False,
       'expected': [CT(['a', 'b']), 'c', 'd'],
       'paths': [(0,), (1,), (2,)]},
      {'s1': [CT([1, 2, 3])],
       's2': [5],
       'expand_composites': False,
       'expected': [5],
       'paths': [(0,)]},
      {'s1': [[CT([9, 9, 9])], 999, {'y': CT([9, 9])}],
       's2': [[CT([1, 2, 3])], 100, {'y': CT([CT([4, 5]), 6])}],
       'expand_composites': False,
       'expected': [CT([1, 2, 3]), 100, CT([CT([4, 5]), 6])],
       'paths': [(0, 0), (1,), (2, 'y')]},
      {'s1': [[CT([9, 9, 9])], 999, {'y': CT([CT([9, 9]), 9])}],
       's2': [[CT([1, 2, 3])], 100, {'y': CT([5, 6])}],
       'expand_composites': False,
       'expected': [CT([1, 2, 3]), 100, CT([5, 6])],
       'paths': [(0, 0), (1,), (2, 'y')]},
  ])  # pyformat: disable
  def testNestFlattenUpTo(self, s1, s2, expected, paths,
                          expand_composites=True):
    result = nest.flatten_up_to(s1, s2, expand_composites=expand_
    self.assertEqual(expected, result)

    result_with_paths = nest.flatten_with_tuple_paths_up_to(
        s1, s2, expand_composites=expand_composites)
    self.assertEqual(result_with_paths, list(zip(paths, expected)

  @parameterized.parameters([
      {'structure': CT(0),
       'sequence': [5],
       'expected': CT(5)},
      {'structure': CT(['a', 'b', 'c']),
       'sequence': ['A', CT(['b']), {'x': 'y'}],
       'expected': CT(['A', CT(['b']), {'x': 'y'}])},
      {'structure': [{'k1': CT('a')}, CT(['b', {'x': CT({'y':
       'sequence': ['A', 'B', 'C'],
       'expected': [{'k1': CT('A')}, CT(['B', {'x': CT({'y
      {'structure': [{'k1': CT('a')}, CT(['b', {'x': CT
       'sequence': ['A', 'B'],
       'expand_composites': False,
       'expected': [{'k1': 'A'}, 'B']},
      {'structure': CT(0, metadata='abc'),
       'sequence': [5],
       'expected': CT(5, metadata='abc')},
  ])  # pyformat: disable
  def testNestPackSequenceAs(self,
                             structure,
                             sequence,
                             expected,
                             expand_composites=True):
    result = nest.pack_sequence_as(
        structure, sequence, expand_composites=expand_composites)
    self.assertEqual(result, expected)

  @parameterized.parameters([
      {'s1': CT('abc'), 's2': CT('xyz')},
      {'s1': CT(['a', 'b', 'c']), 's2': CT(['d', 'e', 'f'])},
      {'s1': [1, CT([10]), CT(200, metadata='xyz')],
       's2': [8, CT([55]), CT(100, metadata='xyz')]},
  ])  # pyformat: disable
  def testNestAssertSameStructure(self, s1, s2, expand_composite
    nest.assert_same_structure(s1, s2, expand_composites=expand_
    nest.assert_shallow_structure(s1, s2, expand_composites=expar

  @parameterized.parameters([
      {'s1': CT(0), 's2': CT(['x'])},
      {'s1': CT([1]), 's2': CT([1, 2])},
      {'s1': CT({'x': 1}), 's2': CT({'y': 1})},
      {'s1': CT(0), 's2': CT(0, metadata='xyz')},
      {'s1': CT(0, metadata='xyz'), 's2': CT(0)},
      {'s1': CT(0, metadata='xyz'), 's2': CT(0, metadata='abc')},
      {'s1': CT(['a', 'b', 'c']), 's2': CT(['d', 'e'])},
      {'s1': [1, CT(['a']), CT('b', metadata='xyz')],
```

```python
       's2': [8, CT([55, 66]), CT(100, metadata='abc')]},
      {'s1': CT(0), 's2': CT2(0), 'error': TypeError},
  ])  # pyformat: disable
  def testNestAssertSameStructureCompositeMismatch(self,
                                                   s1,
                                                   s2,
                                                   error=ValueEr
    # s1 and s2 have the same structure if expand_composites=Fa
    # different structures if expand_composites=True.
    nest.assert_same_structure(s1, s2, expand_composites=False)
    nest.assert_shallow_structure(s1, s2, expand_composites=Fal
    with self.assertRaises(error):  # pylint: disable=g-error-pr
      nest.assert_same_structure(s1, s2, expand_composites=True

  @parameterized.parameters([
      # Note: there are additional test cases in testNestAssert
      {'s1': [1], 's2': [CT(1)]},
      {'s1': [[CT([1, 2, 3])], 100, {'y': CT([5, 6])}],
       's2': [[CT([1, 2, 3])], 100, {'y': CT([CT([4, 5]), 6])}]
       'expand_composites': False},
      {'s1': [[CT([1, 2, 3])], 100, {'y': CT([CT([4, 5]), 6])}]
       's2': [[CT([1, 2, 3])], 100, {'y': CT([5, 6])}],
       'expand_composites': False},
  ])  # pyformat: disable
  def testNestAssertShallowStructure(self, s1, s2, expand_compo
    nest.assert_shallow_structure(s1, s2, expand_composites=exp

  @parameterized.parameters([
          there are additi   test cases in
                           eCompositeMismatch.
                      100,  'y': CT([CT([4, 5]), 6])}]
                      10     y': CT([5, 6])}]},



                    ctur  mpositeMismatch(self,
                                          s1,
                                          s2,
                                          check_type
                       , ValueError)):  # pylint:

                       es=True, check_types=check_type

                 ers([
      {'structure': CT(1, metadata=2),
       'expected': CT(11, metadata=2)},
      {'structure': CT({'x': 1, 'y': [2, 3]}, metadata=2),
       'expected': CT({'x': 11, 'y': [12, 13]}, metadata=2)},
      {'structure': [[CT([1, 2, 3])], 100, {'y': CT([CT([4, 5])
       'expected': [[CT([11, 12, 13])], 110, {'y': CT([CT([14, 1
  ])  # pyformat: disable
  def testNestMapStructure(self, structure, expected, expand_cor
    func = lambda x: x + 10
    result = nest.map_structure(
        func, structure, expand_composites=expand_composites)
    self.assertEqual(result, expected)

  @parameterized.parameters([
      {'s1': [[CT([1, 2, 3])], 100, {'y': 4}],
       's2': [[CT([1, 2, 3])], 100, {'y': CT([CT([4, 5]), 6])}]
       'expected': [[CT([11, 12, 13])], 110, {'y': CT([CT([4, 5
  ])  # pyformat: disable
  def testNestMapStructureUpTo(self, s1, s2, expected):
    func = lambda x: x + 10 if isinstance(x, int) else x
    result = nest.map_structure_up_to(s1, func, s2, expand_compo
    self.assertEqual(result, expected)

  @parameterized.parameters([
      {'structure': CT('a'),
       'expected': CT('CT:a')},
      {'structure': CT(['a', 'b']),
       'expected': CT(['CT/0:a', 'CT/1:b'])},
```

```python
    result = nest.map_structure(
        func, structure, expand_composites=expand_composites)
    self.assertEqual(result, expected)

  @parameterized.parameters([
      {'s1': [[CT([1, 2, 3])], 100, {'y': 4}],
       's2': [[CT([1, 2, 3])], 100, {'y': CT([CT([4, 5]), 6])}],
       'expected': [[CT([11, 12, 13])], 110, {'y': CT([CT([4, 5]), 6])}]}
  ])  # pyformat: disable
  def testNestMapStructureUpTo(self, s1, s2, expected):
    func = lambda x: x + 10 if isinstance(x, int) else x
    result = nest.map_structure_up_to(s1, func, s2, expand_composites=True)
    self.assertEqual(result, expected)

  @parameterized.parameters([
      {'structure': CT('a'),
       'expected': CT('CT:a')},
      {'structure': CT(['a', 'b']),
       'expected': CT(['CT/0:a', 'CT/1:b'])},
    refs = []
    for ct in [ct1, ct2, ct3, ct4]:
      refs.append(weakref.ref(ct))
      refs.append(weakref.ref(ct.components))
      refs.append(weakref.ref(ct.metadata))
    del ct  # pylint: disable=undefined-loop-variable

    for ref in refs:
      self.assertIsNotNone(ref())

    del ct1, ct2, ct3, ct4
    gc.collect()
    for ref in refs:
      self.assertIsNone(ref())

  # pylint: disable=g-long-lambda
  @parameterized.named_parameters([
      ('IndexedSlicesNoDenseShape', lambda: ops.IndexedSlices(
          constant_op.constant([1, 2, 3]), constant_op.constant([2, 8, 4]))),
      ('IndexedSlicesInt32DenseShape', lambda: ops.IndexedSlices(
          constant_op.constant([1, 2, 3]), constant_op.constant([2, 8, 4]),
          constant_op.constant([10], dtypes.int32))),
      ('IndexedSlicesInt64DenseShape', lambda: ops.IndexedSlices(
          constant_op.constant([[1, 2], [3, 4]]), constant_op.constant([2, 8]),
          constant_op.constant([10, 2], dtypes.int64))),
      ('RaggedTensorRaggedRank1',
       lambda: ragged_factory_ops.constant([[1, 2], [3]])),
      ('RaggedTensorRaggedRank2',
       lambda: ragged_factory_ops.constant([[[1, 2], [3]], [[6, 7, 8]]])),
      ('SparseTensor',
       lambda: sparse_tensor.SparseTensor([[3], [7]], ['a', 'b'], [10])),
      ('Nested structure', lambda: {
          'a':
              ops.IndexedSlices(
                  constant_op.constant([1, 2, 3]),
                  constant_op.constant([2, 8, 4])),
          'b': [
              ragged_factory_ops.constant([[1, 2], [3]]),
              sparse_tensor.SparseTensor([[3], [7]], ['a', 'b'], [10])
          ]
      }),
  ])
  def testAssertSameStructureWithValueAndTypeSpec(self, value_func):
    value = value_func()
    spec = nest.map_structure(type_spec.type_spec_from_value, value,
                              expand_composites=False)
    nest.assert_same_structure(value, spec, expand_composites=True)


if __name__ == '__main__':
  googletest.main()
```



```
    return self._type_spec_class(component_specs, self.met

  def __repr__(self):
    return '%s(%r, %r)' % (type(self).__name__, self.compo

  def __eq__(self, other):
    return (type(self) is type(other) and
            self.components == other.components and
            self.metadata == other.metadata)


# Another test CompositeTensor class.  `tf.nest` should tr
# classes as different structure types (e.g. for assert_sa
class CTSpec2(CTSpec):
  pass


class CT2(CT):
  _type_spec_class = CTSpec2


@test_util.run_all_in_graph_and_eager_modes
class CompositeTensorTest(test_util.TensorFlowTestCase, pa

  @parameterized.parameters([
      {'structure': CT(0),
       'expected': [0],
       'paths': [('CT',)]},
      {'structure': CT('a'),
       'expected': ['a'],
       'paths': [('CT',)]},
      {'structure': CT(['a', 'b', 'c']),
       'expected': ['a', 'b', 'c'],
       'paths': [('CT', 0), ('CT', 1), ('CT', 2)]},
      {'structure': CT({'x': 'a', 'y': 'b', 'z': 'c'}),
       'expected': ['a', 'b', 'c'],
       'paths': [('CT', 'x'), ('CT', 'y'), ('CT', 'z')]},
      {'structure': [{'k1': CT('a')}, CT(['b', {'x': CT({
       'expected': ['a', 'b', 'c'],
       'paths': [(0, 'k1', 'CT'), (1, 'CT', 0), (1, 'CT',
      {'structure': CT(0),
       'expand_composites': False,
       'expected': [CT(0)],
       'paths': [()]},
      {'structure': [{'k1': CT('a')}, CT(['b', {'x': CT({
       'expand_composites': False,
       'expected': [CT('a'), CT(['b', {'x': CT({'y': 'c'})
       'paths': [(0, 'k1'), (1,)]},
  ])  # pyformat: disable
  def testNestFlatten(self, structure, expected, paths, ex
    result = nest.flatten(structure, expand_composites=exp
    self.assertEqual(result, expected)

    result_with_paths = nest.flatten_with_tuple_paths(
        structure, expand_composites=expand_composites)
    self.assertEqual(result_with_paths, list(zip(paths, ex

    string_paths = ['/'.join(str(p) for p in path) for pat
    result_with_string_paths = nest.flatten_with_joined_st
        structure, expand_composites=expand_composites)
    self.    ertEqual(r    lt_with_string_paths,
                     l    (zip(string_paths, expected)))

    flat_p    hs_result     ist(
        ne    yield_fla    aths(st             and_composites
    self.a    tEqual(    t_pa    s_resul     pa

  @parameterized.parameters([
      {'s1': [1, 2, 3],
       's2': [CT(['a', 'b']), 'c', 'd'],
       'expand_composites': False,
       'expected': [CT(['a', 'b']), 'c', 'd'],
       'paths': [(0,), (1,), (2,)]},
```

```
       'expand_composites': False,
       'expected': [CT(['a', 'b']), 'c', 'd'],
       'paths': [(0,), (1,), (2,)]},
      {'s1': [CT([1, 2, 3])],
       's2': [5],
       'expand_composites': False,
       'expected': [5],
       'paths': [(0,)]},
      {'s1': [[CT([9, 9, 9])], 999, {'y': CT([9, 9])}],
       's2': [[CT([1, 2, 3])], 100, {'y': CT([CT([4, 5]), 6])}],
       'expand_composites': False,
       'expected': [CT([1, 2, 3]), 100, CT([CT([4, 5]), 6])],
       'paths': [(0, 0), (1,), (2, 'y')]},
      {'s1': [[CT([9, 9, 9])], 999, {'y': CT([CT([9, 9]), 9])}],
       's2': [[CT([1, 2, 3])], 100, {'y': CT([5, 6])}],
       'expand_composites': False,
       'expected': [CT([1, 2, 3]), 100, CT([5, 6])],
       'paths': [(0, 0), (1,), (2, 'y')]},
  ])  # pyformat: disable
  def testNestFlattenUpTo(self, s1, s2, expected, paths,
                          expand_composites=True):
    result = nest.flatten_up_to(s1, s2, expand_composites=expand_
    self.assertEqual(expected, result)

    result_with_paths = nest.flatten_with_tuple_paths_up_to(
        s1, s2, expand_composites=expand_composites)
    self.assertEqual(result_with_paths, list(zip(paths, expected)

  @parameterized.parameters([
      {'structure': CT(0),
       'sequence': [5],
       'expected': CT(5)},
      {'structure': CT(['a', 'b', 'c']),
       'sequence': ['A', CT(['b']), {'x': 'y'}],
       'expected': CT(['A', CT(['b']), {'x': 'y'}])},
      {'structure': [{'k1': CT('a')}, CT(['b', {'x': CT({'y': '
       'sequence': ['A', 'B', 'C'],
       'expected': [{'k1': CT('A')}, CT(['B', {'x': CT({'y
      {'structure': [{'k1': CT('a')}, CT(['b', {'x': CT({'y
       'sequence': ['A', 'B'],
       'expand_composites': False,
       'expected': [{'k1': 'A'}, 'B']},
      {'structure': CT(0, metadata='abc'),
       'sequence': [5],
       'expected': CT(5, metadata='abc')},
  ])  # pyformat: disable
  def testNestPackSequenceAs(self,
                             structure,
                             sequence,
                             expected,
                             expand_composites=True):
    result = nest.pack_sequence_as(
        structure, sequence, expand_composites=expand_composites)
    self.assertEqual(result, expected)

  @parameterized.parameters([
      {'s1': CT('abc'), 's2': CT('xyz')},
      {'s1': CT(['a', 'b', 'c']), 's2': CT(['d', 'e', 'f'])},
      {'s1': [1, CT([10]), CT(200, metadata='xyz')],
       's2': [8, CT([55]), CT(100, metadata='xyz')]},
  ])  # pyformat: disable
  def testNest    tSameStructure(self, s1, s2, exp    d_composite
    t.asse    e_st          s2,     ompo          nd_c
    t.ass    allow_structu    s1,    , expa    cosites    pa

  @para    ized    rameters([
      {'s     : CT(       's2     }),
      {'s1': CT([1]), 's2': CT([1, 2])},
      {'s1': CT({'x': 1}), 's2': CT({'y': 1})},
      {'s1': CT(0), 's2': CT(0, metadata='xyz')},
      {'s1': CT(0, metadata='xyz'), 's2': CT(0)},
      {'s1': CT(0, metadata='xyz'), 's2': CT(0, metadata='abc')},
      {'s1': CT(['a', 'b', 'c']), 's2': CT(['d', 'e'])},
      {'s1': [1, CT(['a']), CT('b', metadata='xyz')],
```

```
       's2': [8, CT([55, 66]), CT(100, metadata='abc')]},
      {'s1': CT(0), 's2': CT2(0), 'error': TypeError},
  ])  # pyformat: disable
  def testNestAssertSameStructureCompositeMismatch(self,
                                                   s1,
                                                   s2,
                                                   error=ValueE
    # s1 and s2 have the same structure if expand_composites=Fa
    # different structures if expand_composites=True.
    nest.assert_same_structure(s1, s2, expand_composites=False)
    nest.assert_shallow_structure(s1, s2, expand_composites=Fal
    with self.assertRaises(error):  # pylint: disable=g-error-pr
      nest.assert_same_structure(s1, s2, expand_composites=True

  @parameterized.parameters([
      # Note: there are additional test cases in testNestAssert
      {'s1': [1], 's2': [CT(1)]},
      {'s1': [[CT([1, 2, 3])], 100, {'y': CT([5, 6])}],
       's2': [[CT([1, 2, 3])], 100, {'y': CT([CT([4, 5]), 6])}]
       'expand_composites': False},
      {'s1': [[CT([1, 2, 3])], 100, {'y': CT([CT([4, 5]), 6])}]
       's2': [[CT([1, 2, 3])], 100, {'y': CT([5, 6])}],
       'expand_composites': False},
  ])  # pyformat: disable
  def testNestAssertShallowStructure(self, s1, s2, expand_compo
    nest.assert_shallow_structure(s1, s2, expand_composites=exp

  @parameterized.parameters([
             there are addit      test cases in
                              CompositeMismatch.
                         100, {'y': CT([CT([4, 5]), 6])}]
                         100, {'y': CT([5, 6])}]},

                 ctu    ompositeMismatch(self,
                                          s1,
                                          s2,
                                          check_typ
                               , ValueError)):  # pylint:

                   es=True, check_types=check_typ
              ers([
      {'structure': CT(1, metadata=2),
       'expected': CT(11, metadata=2)},
      {'structure': CT({'x': 1, 'y': [2, 3]}, metadata=2),
       'expected': CT({'x': 11, 'y': [12, 13]}, metadata=2)},
      {'structure': [[CT([1, 2, 3])], 100, {'y': CT([CT([4, 5]
       'expected': [[CT([11, 12, 13])], 110, {'y': CT([CT([14, 
  ])  # pyformat: disable
  def testNestMapStructure(self, structure, expected, expand_co
    func = lambda x: x + 10
    result = nest.map_structure(
        func, structure, expand_compos    s=expand_composites)
    self.    ertE    l(res    , expected)

  @parame    ize    arameters([
      {'s    [C    ]], 100, {'
       's2': [[CT([1, 2, 3])], 100, {'y': CT([CT([4, 5]), 6])}]
       'expected': [[CT([11, 12, 13])], 110, {'y': CT([CT([4, 5
  ])  # pyformat: disable
  def testNestMapStructureUpTo(self, s1, s2, expected):
    func = lambda x: x + 10 if isinstance(x, int) else x
    result = nest.map_structure_up_to(s1, func, s2, expand_comp
    self.assertEqual(result, expected)

  @parameterized.parameters([
      {'structure': CT('a'),
       'expected': CT('CT:a')},
      {'structure': CT(['a', 'b']),
       'expected': CT(['CT/0:a', 'CT/1:b'])},
```

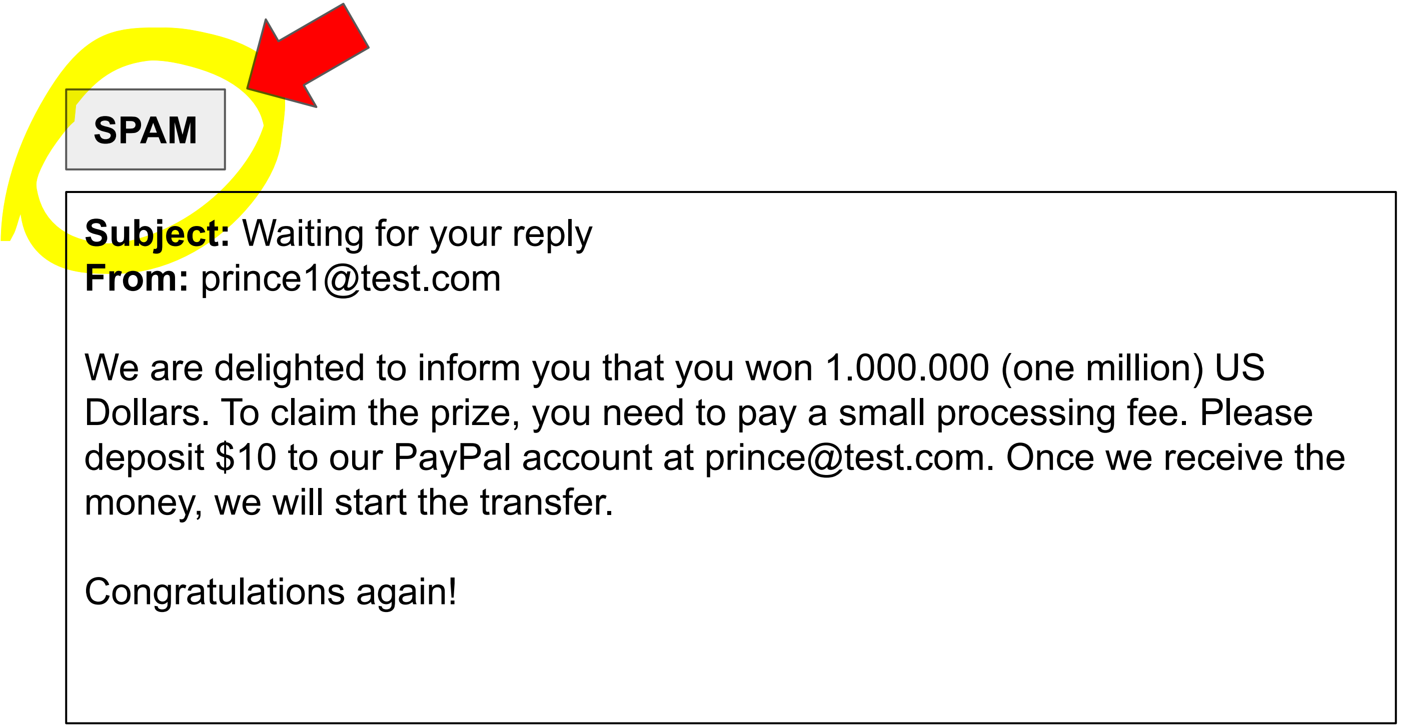
```
    result = nest.map_structure(
        func, structure, expand_composites=expand_composites)
    self.assertEqual(result, expected)

  @parameterized.parameters([
      {'s1': [[CT([1, 2, 3])], 100, {'y': 4}],
       's2': [[CT([1, 2, 3])], 100, {'y': CT([CT([4, 5]), 6])}],
       'expected': [[CT([11, 12, 13])], 110, {'y': CT([CT([4, 5]), 6])}]}
  ])  # pyformat: disable
  def testNestMapStructureUpTo(self, s1, s2, expected):
    func = lambda x: x + 10 if isinstance(x, int) else x
    result = nest.map_structure_up_to(s1, func, s2, expand_composites=True)
    self.assertEqual(result, expected)

  @parameterized.parameters([
      {'structure': CT('a'),
       'expected': CT('CT:a')},
      {'structure': CT(['a', 'b']),
       'expected': CT(['CT/0:a', 'CT/1:b'])},

    refs = []
    for ct in [ct1, ct2, ct3, ct4]:
      refs.append(weakref.ref(ct))
      refs.append(weakref.ref(ct.components))
      refs.append(weakref.ref(ct.metadata))
    del ct  # pylint: disable=undefined-loop-variable

    for ref in refs:
      self.assertIsNotNone(ref())

    del ct1, ct2, ct3, ct4
    gc.collect()
    for ref in refs:
      self.assertIsNone(ref())

  # pylint: disable=g-long-lambda
  @parameterized.named_parameters([
      ('IndexedSlicesNoDenseShape', lambda: ops.IndexedSlices(
          constant_op.constant([1, 2, 3]), constant_op.constant([2, 8, 4]))),
      ('IndexedSlicesInt32DenseShape', lambda: ops.IndexedSlices(
          constant_op.constant([1, 2, 3]), constant_op.constant([2, 8, 4]),
          constant_op.constant([10], dtypes.int32))),
      ('IndexedSlicesInt64DenseShape', lambda: ops.IndexedSlices(
          constant_op.constant([[1, 2], [3, 4]]), constant_op.constant([2, 8]),
          constant_op.constant([10, 2], dtypes.int64))),
      ('RaggedTensorRaggedRank1',
       lambda: ragged_factory_ops.constant([[1, 2], [3]])),
      ('RaggedTensorRaggedRank2',
       lambda: ragged_factory_ops.constant([[[1, 2], [3]], [[6, 7, 8]]])),
      ('SparseTensor',
       lambda: sparse_tensor.SparseTensor([[3], [7]], ['a', 'b'], [10])),
      ('Nested structure', lambda: {
          'a':
              ops.IndexedSlices(
                  constant_op.constant([1, 2, 3]),
                  constant_op.constant([2, 8, 4])),
          'b
              agged_    tory_op    cons    nt(    2],    ]),
              parse_    sor.Sp    seTe    r([[    , [7]],    'a'    'b'],
          ]
      }),
  ])
  def testAssertSameStructureWithValueAndTypeSpec(self, value_func):
    value = value_func()
    spec = nest.map_structure(type_spec.type_spec_from_value, value,
                              expand_composites=False)
    nest.assert_same_structure(value, spec, expand_composites=True)


if __name__ == '__main__':
  googletest.main()
```

Use Machine Learning!

Machine Learning

- Get data
- Define & calculate features
- Train and use the model

Getting data



Subject: Waiting for your reply
From: prince1@test.com

We are delighted to inform you that you won 1.000.000 (one million) US Dollars. To claim the prize, you need to pay a small processing fee. Please deposit $10 to our PayPal account at prince@test.com. Once we receive the money, we will start the transfer.

Congratulations again!

Machine Learning

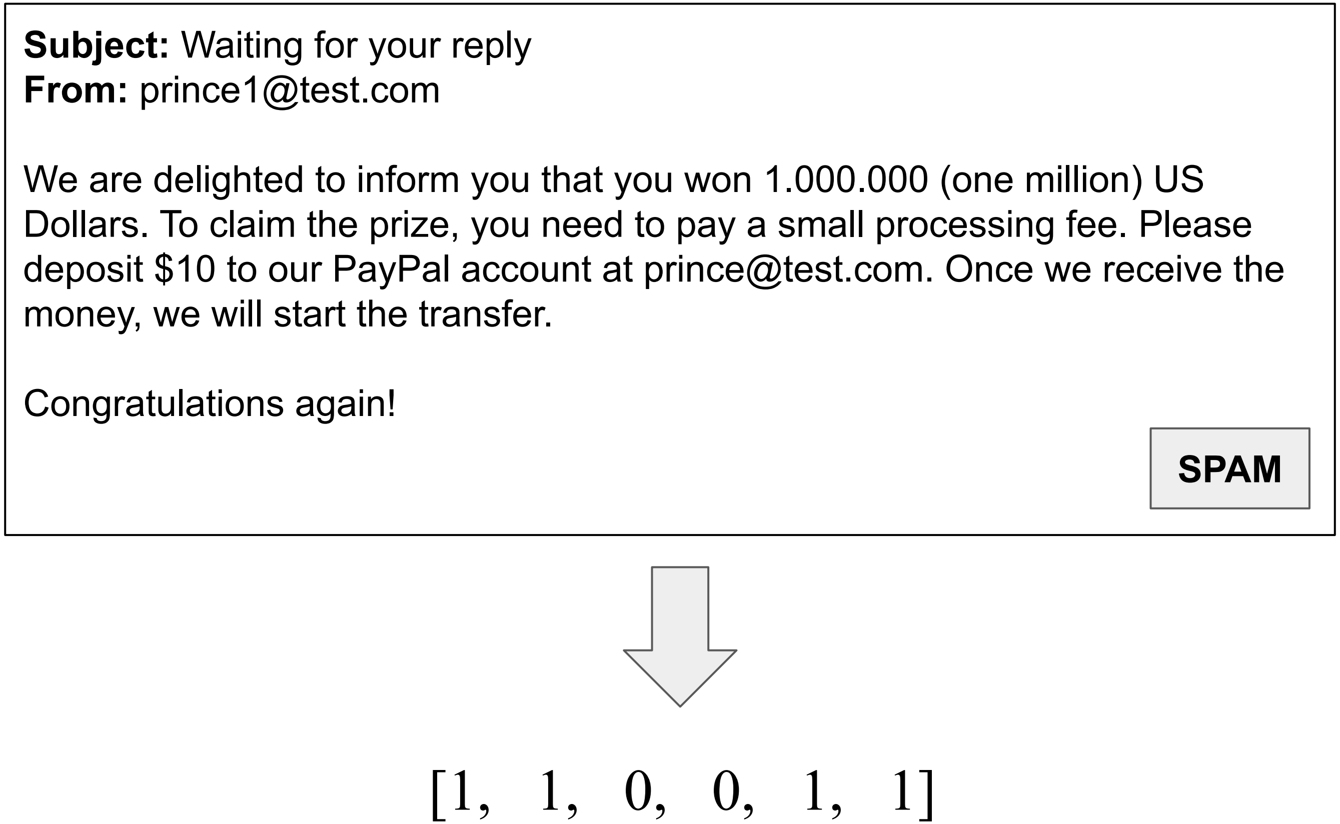
- Get data
- **Define & calculate features**
- Train and use the model

Features

- Length of title > 10? true/false
- Length of body > 10? true/false
- Sender "promotions@online.com"? true/false
- Sender "hpYOSKmL@test.com"? true/false
- Sender domain "test.com"? true/false
- Description contains "deposit"? true/false

Rules

Start with rules and then use these rules as features

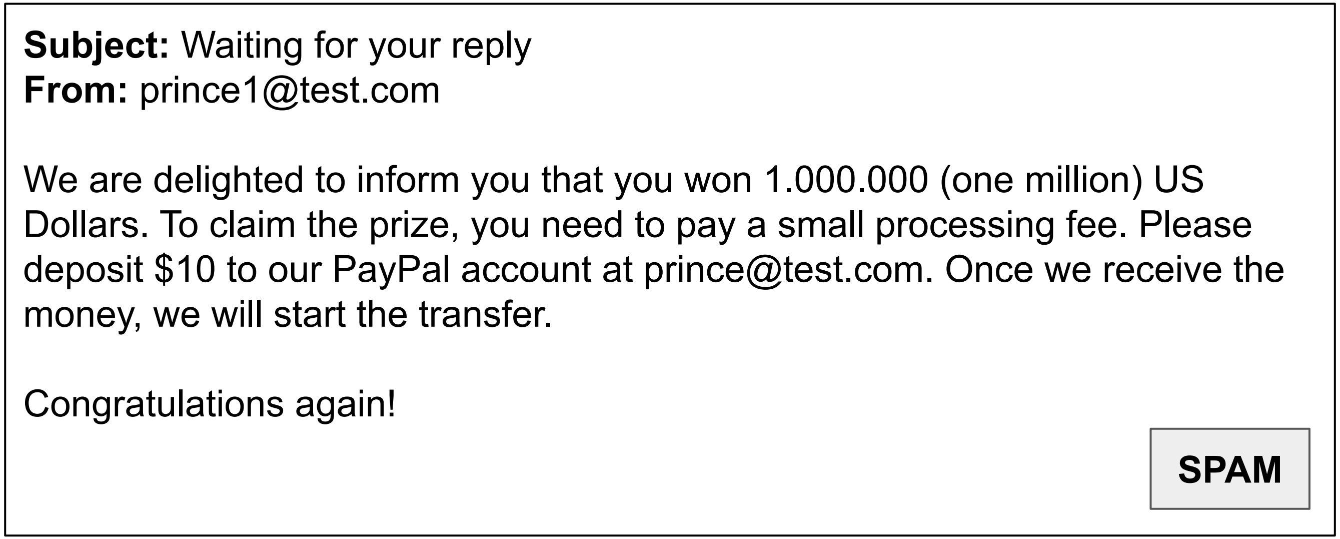


Subject: Waiting for your reply
From: prince1@test.com

We are delighted to inform you that you won 1.000.000 (one million) US Dollars. To claim the prize, you need to pay a small processing fee. Please deposit $10 to our PayPal account at prince@test.com. Once we receive the money, we will start the transfer.

Congratulations again!

$$[1, \quad 1, \quad 0, \quad 0, \quad 1, \quad 1]$$



Subject: Waiting for your reply
From: prince1@test.com

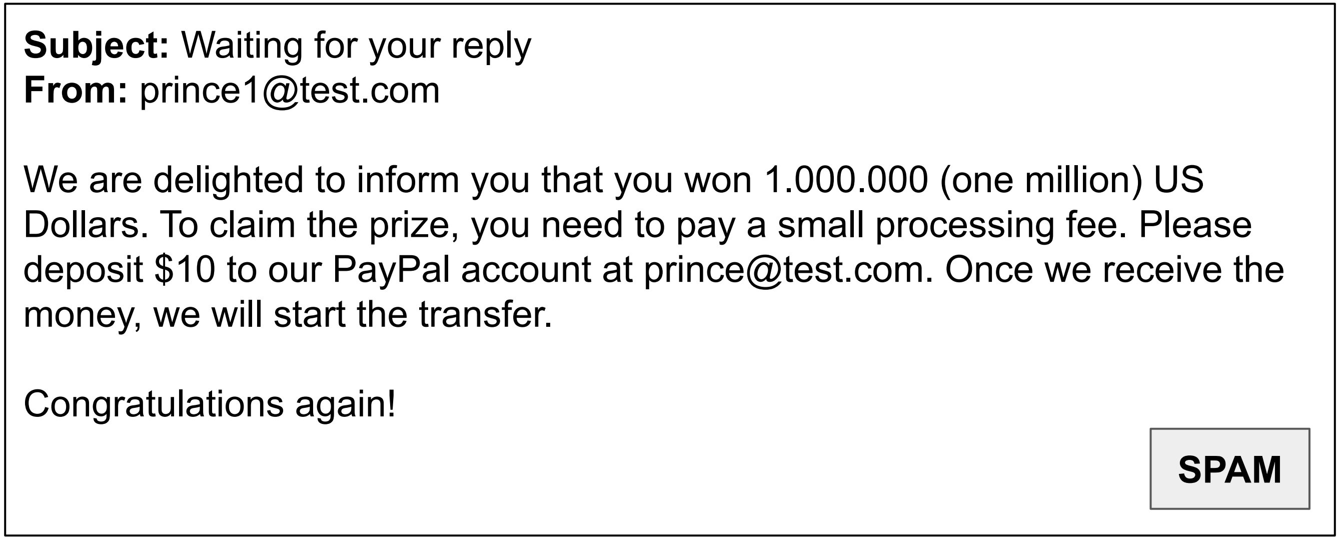
We are delighted to inform you that you won 1.000.000 (one million) US Dollars. To claim the prize, you need to pay a small processing fee. Please deposit $10 to our PayPal account at prince@test.com. Once we receive the money, we will start the transfer.

Congratulations again!

Length of title > 10? **True**

T

$$[1,\ 1,\ 0,\ 0,\ 1,\ 1]$$



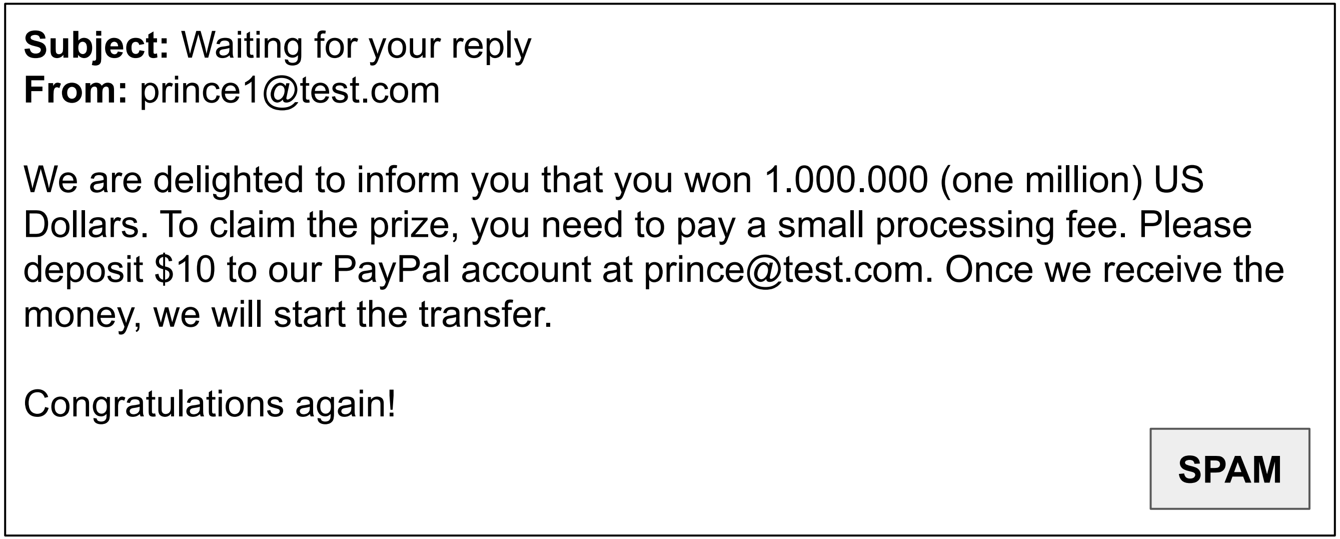
Subject: Waiting for your reply
From: prince1@test.com

We are delighted to inform you that you won 1.000.000 (one million) US Dollars. To claim the prize, you need to pay a small processing fee. Please deposit $10 to our PayPal account at prince@test.com. Once we receive the money, we will start the transfer.

Congratulations again!

$$[1,\quad 1,\quad 0,\quad 0,\quad 1,\quad 1]$$

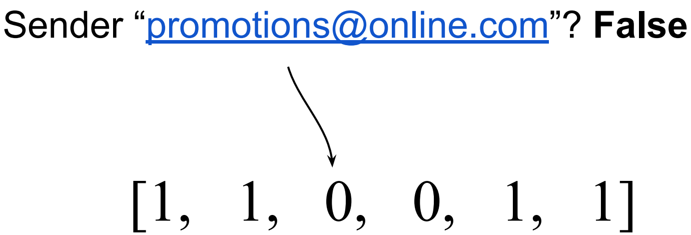
Length of body > 10? **True**



Subject: Waiting for your reply
From: prince1@test.com

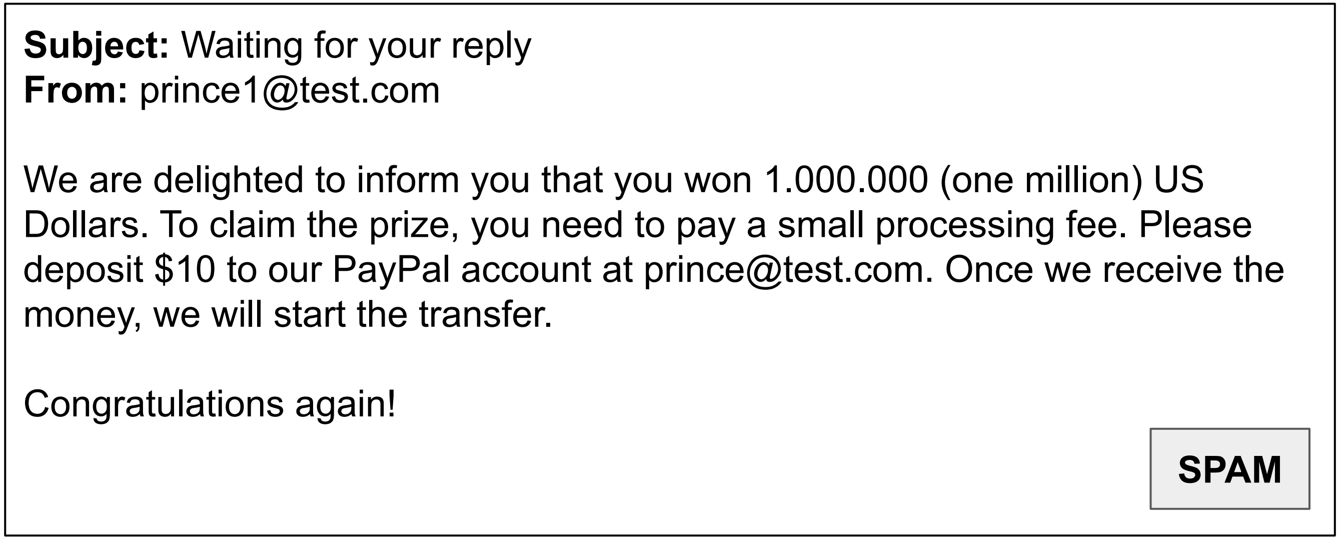
We are delighted to inform you that you won 1.000.000 (one million) US Dollars. To claim the prize, you need to pay a small processing fee. Please deposit $10 to our PayPal account at prince@test.com. Once we receive the money, we will start the transfer.

Congratulations again!



Sender “promotions@online.com”? **False**

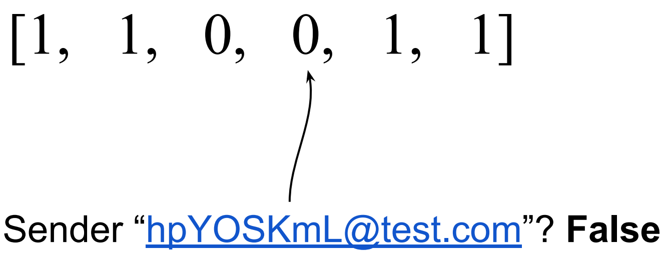
$[1, \; 1, \; 0, \; 0, \; 1, \; 1]$



Subject: Waiting for your reply
From: prince1@test.com

We are delighted to inform you that you won 1.000.000 (one million) US Dollars. To claim the prize, you need to pay a small processing fee. Please deposit $10 to our PayPal account at prince@test.com. Once we receive the money, we will start the transfer.

Congratulations again!



$$[1, \quad 1, \quad 0, \quad 0, \quad 1, \quad 1]$$

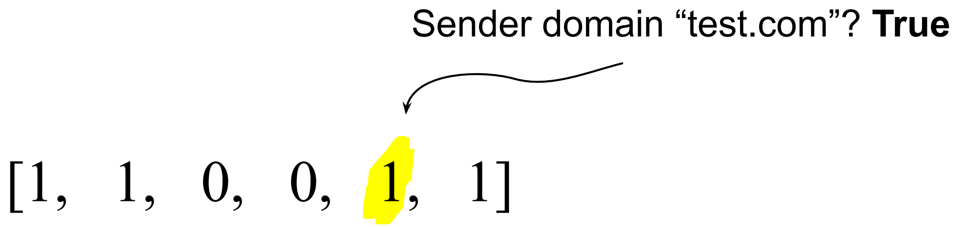
Sender “hpYOSKmL@test.com”? **False**

Subject: Waiting for your reply
From: prince1@test.com

We are delighted to inform you that you won 1.000.000 (one million) US Dollars. To claim the prize, you need to pay a small processing fee. Please deposit $10 to our PayPal account at prince@test.com. Once we receive the money, we will start the transfer.

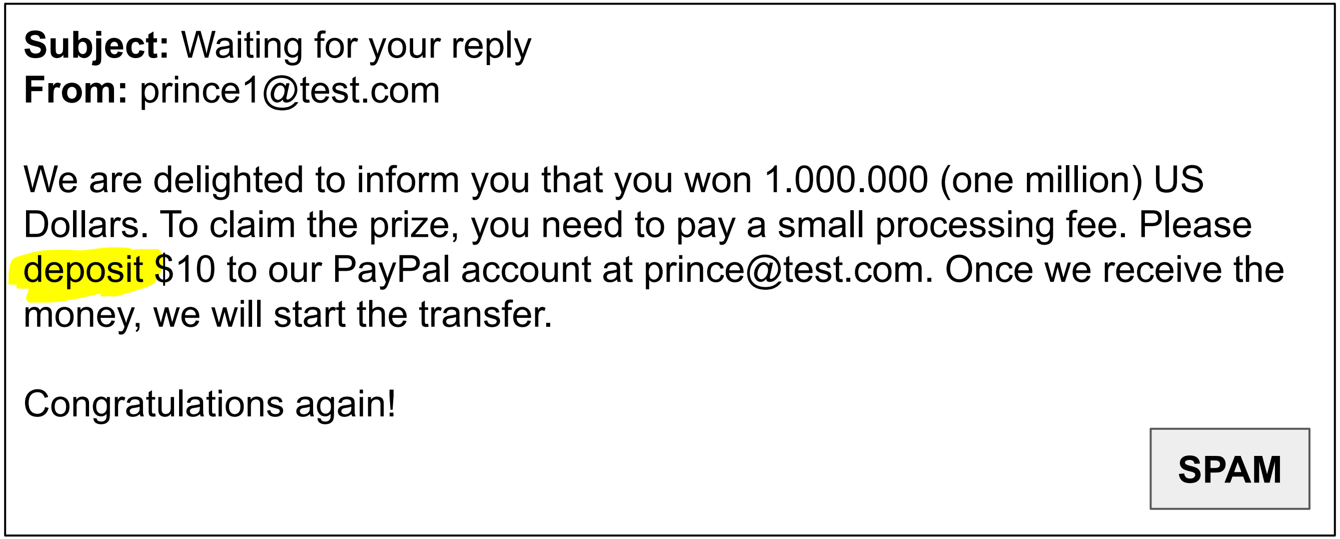
Congratulations again!

SPAM



Sender domain “test.com”? **True**

[1, 1, 0, 0, 1, 1]



Subject: Waiting for your reply
From: prince1@test.com

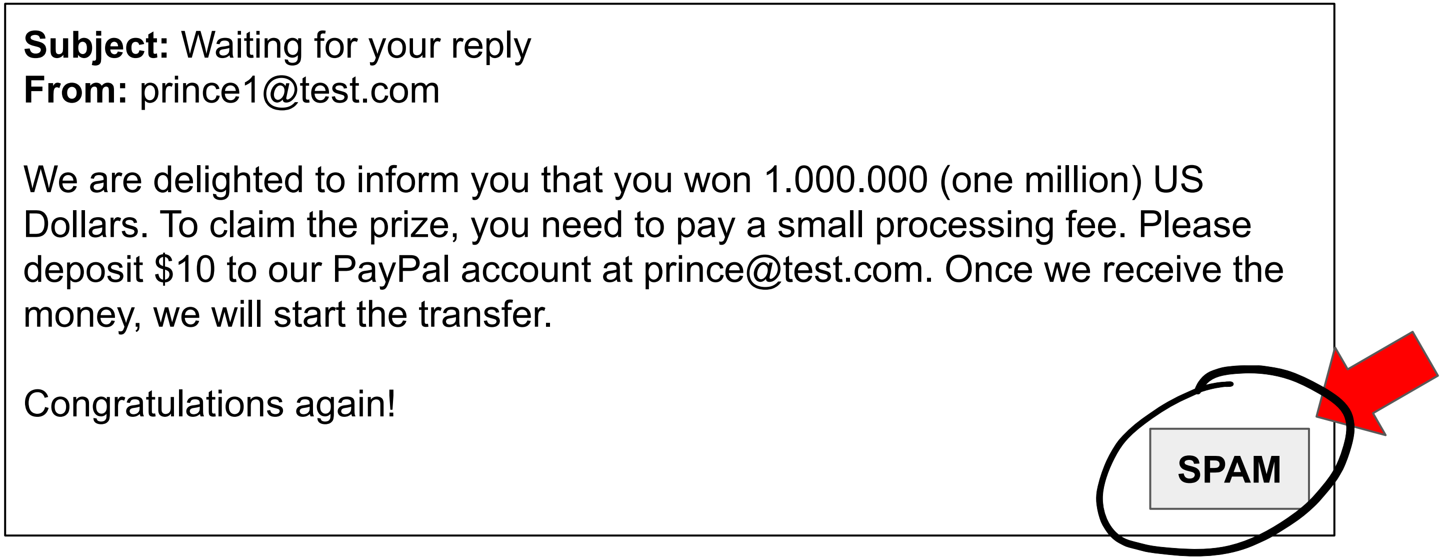
We are delighted to inform you that you won 1.000.000 (one million) US Dollars. To claim the prize, you need to pay a small processing fee. Please deposit $10 to our PayPal account at prince@test.com. Once we receive the money, we will start the transfer.

Congratulations again!

$$[1, \quad 1, \quad 0, \quad 0, \quad 1, \quad 1]$$

TRUE

Description contains “deposit”? ~~**False**~~



Subject: Waiting for your reply
From: prince1@test.com

We are delighted to inform you that you won 1.000.000 (one million) US Dollars. To claim the prize, you need to pay a small processing fee. Please deposit $10 to our PayPal account at prince@test.com. Once we receive the money, we will start the transfer.

Congratulations again!

$[1, \ 1, \ 0, \ 0, \ 1, \ 1]$ $\qquad$ 1

Features (data)	Target (desired output)
[1, 1, 0, 0, 1, 1]	1

Features (data)	Target (desired output)
[1, 1, 0, 0, 1, 1]	1
[0, 0, 0, 1, 0, 1]	0

Features (data)	Target (desired output)
[1, 1, 0, 0, 1, 1]	1
[0, 0, 0, 1, 0, 1]	0
[1, 1, 1, 0, 1, 0]	1

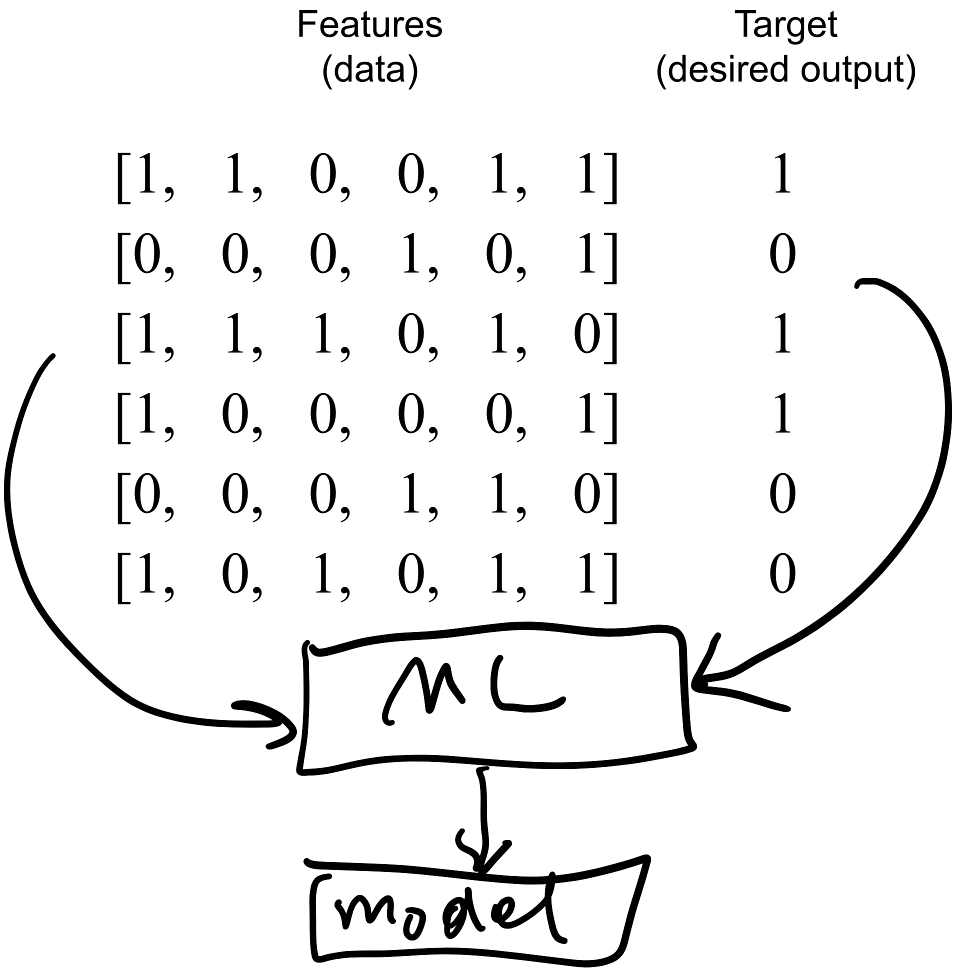
Features (data)	Target (desired output)
[1, 1, 0, 0, 1, 1]	1
[0, 0, 0, 1, 0, 1]	0
[1, 1, 1, 0, 1, 0]	1
[1, 0, 0, 0, 0, 1]	1

Features (data)	Target (desired output)
[1, 1, 0, 0, 1, 1]	1
[0, 0, 0, 1, 0, 1]	0
[1, 1, 1, 0, 1, 0]	1
[1, 0, 0, 0, 0, 1]	1
[0, 0, 0, 1, 1, 0]	0

Features (data)	Target (desired output)
[1, 1, 0, 0, 1, 1]	1
[0, 0, 0, 1, 0, 1]	0
[1, 1, 1, 0, 1, 0]	1
[1, 0, 0, 0, 0, 1]	1
[0, 0, 0, 1, 1, 0]	0
[1, 0, 1, 0, 1, 1]	0

Machine Learning

- Get data
- Define & calculate features
- **Train and use the model**



Features (data)	Target (desired output)
[1, 1, 0, 0, 1, 1]	1
[0, 0, 0, 1, 0, 1]	0
[1, 1, 1, 0, 1, 0]	1
[1, 0, 0, 0, 0, 1]	1
[0, 0, 0, 1, 1, 0]	0
[1, 0, 1, 0, 1, 1]	0

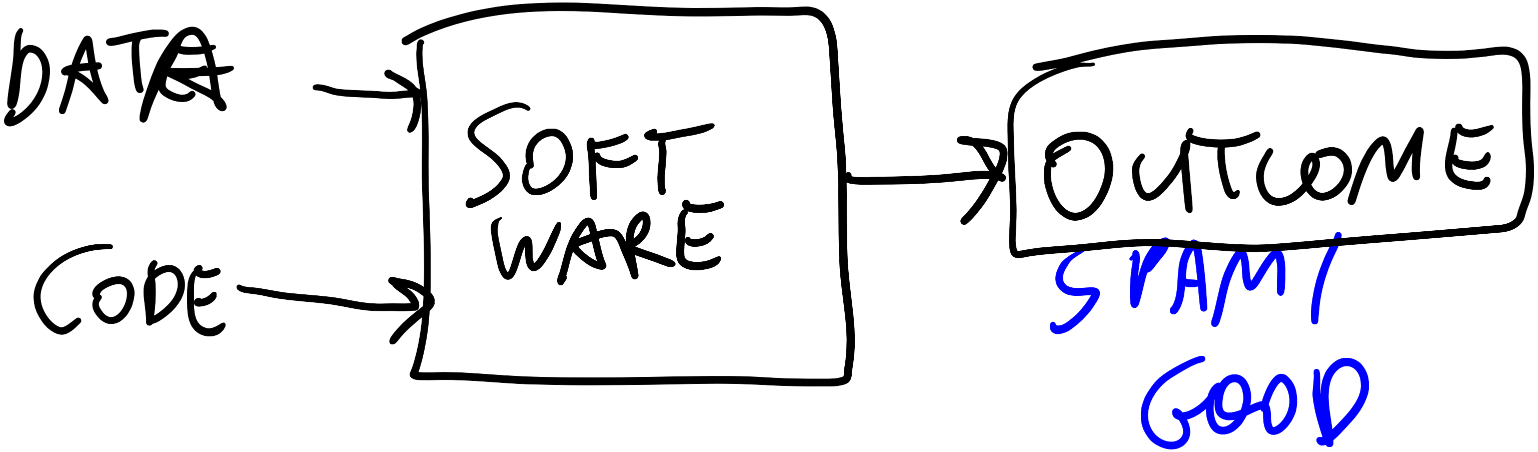
Apply

MODEL

Features (data)	Predictions (output)	Final outcome (decision)
[0, 0, 0, 1, 0, 1]	0.8	SPAM
[0, 0, 0, 1, 1, 0]	0.6	S
[1, 0, 1, 0, 1, 1]	0.1	GOOD
[1, 1, 1, 0, 1, 0]	0.01	G
[1, 0, 0, 0, 0, 1]	0.7	S
[1, 1, 0, 0, 1, 1]	0.4	G

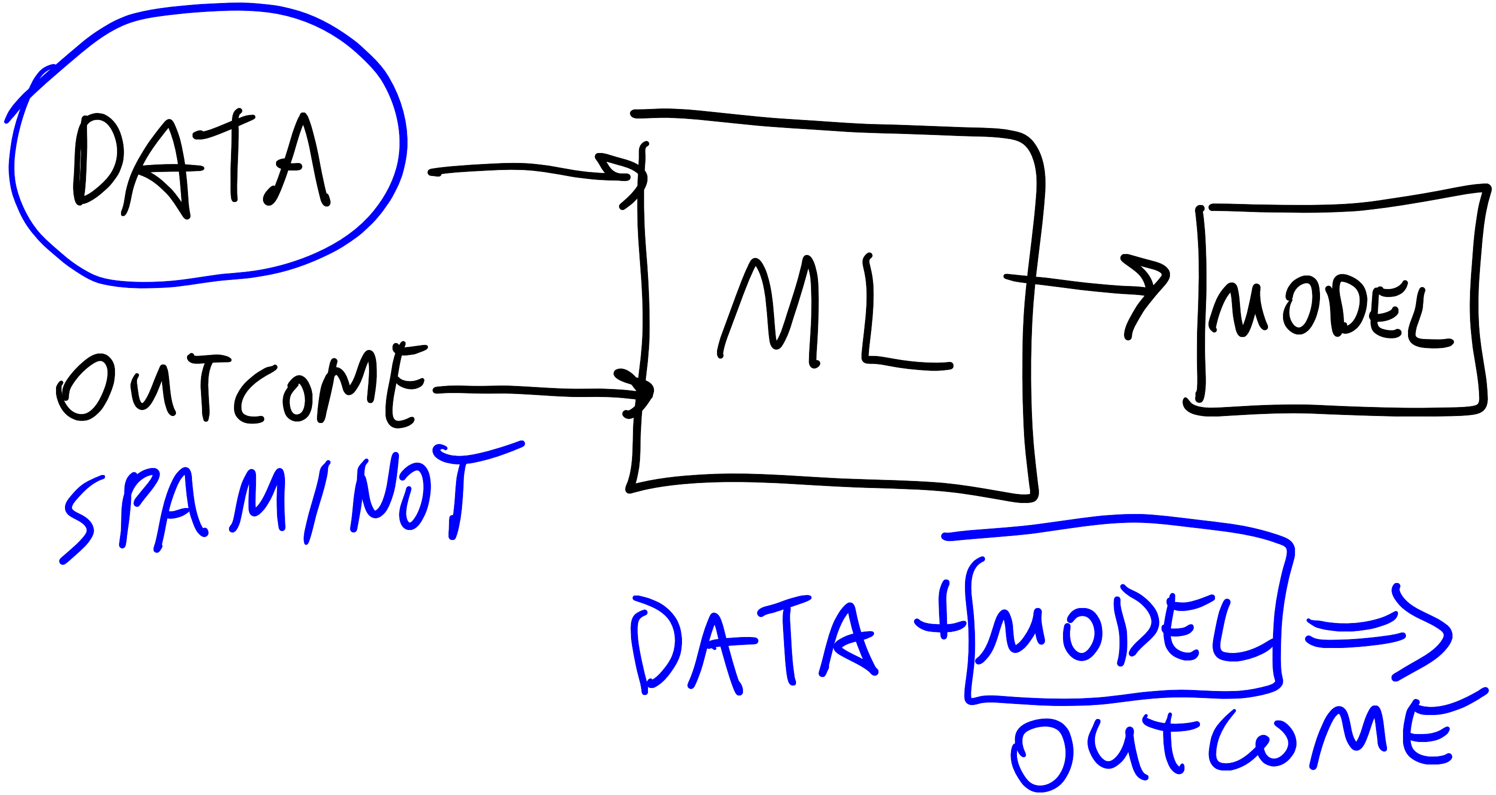
≥ 0.5

Summary



data + code => software => outcome

Summary



data + outcome => ML => model

Next

Supervised machine learning

- A bit more formal definition
- Examples: regression, classification, ranking