You can order print and ebook versions of *Think Python 3e* from and

Work done by Colin Howard starting in Exercises and indicated with boldface text

```python
from os.path import basename, exists


def download(url):
    filename = basename(url)
    if not exists(filename):
        from urllib.request import urlretrieve

        local, _ = urlretrieve(url, filename)
        print("Downloaded " + str(local))
    return filename

download('https://github.com/AllenDowney/ThinkPython/raw/v3/thinkpython.py');
download('https://github.com/AllenDowney/ThinkPython/raw/v3/diagram.py');


import thinkpython
```

```
Downloaded thinkpython.py
Downloaded diagram.py
```

Here are versions of the `Card`, `Deck`, and `Hand` classes from Chapter 17, which we will use in some examples in this chapter.

```python
class Card:
    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
                  '8', '9', '10', 'Jack', 'Queen', 'King', 'Ace']

    def __init__(self, suit, rank):
        self.suit = suit
        self.rank = rank

    def __str__(self):
        rank_name = Card.rank_names[self.rank]
        suit_name = Card.suit_names[self.suit]
        return f'{rank_name} of {suit_name}'
```

```python
import random

class Deck:
    def __init__(self, cards):
        self.cards = cards

    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)

    def make_cards():
```

```python
        cards = []
        for suit in range(4):
            for rank in range(2, 15):
                card = Card(suit, rank)
                cards.append(card)
        return cards

    def shuffle(self):
        random.shuffle(self.cards)

    def pop_card(self):
        return self.cards.pop()

    def add_card(self, card):
        self.cards.append(card)
```

In [4]:
```python
class Hand(Deck):
    def __init__(self, label=''):
        self.label = label
        self.cards = []
```

Python Extras

One of my goals for this book has been to teach you as little Python as possible. When there were two ways to do something, I picked one and avoided mentioning the other. Or sometimes I put the second one into an exercise.

Now I want to go back for some of the good bits that got left behind. Python provides a number of features that are not really necessary -- you can write good code without them -- but with them you can write code that's more concise, readable, or efficient, and sometimes all three.

Sets

Python provides a class called `set` that represents a collection of unique elements. To create an empty set, we can use the class object like a function.

In [5]:
```python
s1 = set()
s1
```

Out[5]:
```
set()
```

We can use the `add` method to add elements.

In [6]:
```python
s1.add('a')
s1.add('b')
s1
```

```
Out[6]:  {'a', 'b'}
```

Or we can pass any kind of sequence to `set`.

```
In [7]:  s2 = set('acd')
         s2
```

```
Out[7]:  {'a', 'c', 'd'}
```

An element can only appear once in a `set`. If you add an element that's already there, it has no effect.

```
In [8]:  s1.add('a')
         s1
```

```
Out[8]:  {'a', 'b'}
```

Or if you create a set with a sequence that contains duplicates, the result contains only unique elements.

```
In [9]:  set('banana')
```

```
Out[9]:  {'a', 'b', 'n'}
```

Some of the exercises in this book can be done concisely and efficiently with sets. For example, here is a solution to an exercise in Chapter 11 that uses a dictionary to check whether there are any duplicate elements in a sequence.

```
In [10]:  def has_duplicates(t):
              d = {}
              for x in t:
                  d[x] = True
              return len(d) < len(t)
```

This version adds the element of `t` as keys in a dictionary, and then checks whether there are fewer keys than elements. Using sets, we can write the same function like this.

```
In [11]:  def has_duplicates(t):
              s = set(t)
              return len(s) < len(t)
```

```
In [12]:  has_duplicates('abba')
```

```
Out[12]:  True
```

An element can only appear in a set once, so if an element in `t` appears more than once, the set will be smaller than `t`. If there are no duplicates, the set will be the same size as `t`.

`set` objects provide methods that perform set operations. For example, `union` computes the union of two sets, which is a new set that contains all elements that appear in either set.

```
In [13]:  s1.union(s2)
```

Out[13]: {'a', 'b', 'c', 'd'}

Some arithmetic operators work with sets. For example, the - operator performs set
subtraction -- the result is a new set that contains all elements from the first set that are *not*
in the second set.

```
In [14]:  s1 - s2
```

Out[14]: {'b'}

In Chapter 12 we used dictionaries to find the words that appear in a document but not in a
word list. We used the following function, which takes two dictionaries and returns a new
dictionary that contains only the keys from the first that don't appear in the second.

```
In [15]:  def subtract(d1, d2):
              res = {}
              for key in d1:
                  if key not in d2:
                      res[key] = d1[key]
              return res
```

With sets, we don't have to write this function ourselves. If `word_counter` is a dictionary
that contains the unique words in the document and `word_list` is a list of valid words, we
can compute the set difference like this.

```
In [16]:  # this cell creates a small example so we can run the following
          # cell without loading the actual data

          word_counter = {'word': 1}
          word_list = ['word']
```

```
In [17]:  set(word_counter) - set(word_list)
```

Out[17]: set()

The result is a set that contains the words in the document that don't appear in the word list.

The relational operators work with sets. For example, <= checks whether one set is a subset
of another, including the possibility that they are equal.

```
In [18]:  set('ab') <= set('abc')
```

Out[18]: True

With these operators, we can use sets to do some of the exercises in Chapter 7. For example,
here's a version of `uses_only` that uses a loop.

```
In [19]: def uses_only(word, available):
             for letter in word:
                 if letter not in available:
                     return False
             return True
```

`uses_only` checks whether all letters in `word` are in `available`. With sets, we can rewrite it like this.

```
In [20]: def uses_only(word, available):
             return set(word) <= set(available)
```

If the letters in `word` are a subset of the letters in `available`, that means that `word` uses only letters in `available`.

Counters

A `Counter` is like a set, except that if an element appears more than once, the `Counter` keeps track of how many times it appears. If you are familiar with the mathematical idea of a "multiset", a `Counter` is a natural way to represent a multiset.

The `Counter` class is defined in a module called `collections`, so you have to import it. Then you can use the class object as a function and pass as an argument a string, list, or any other kind of sequence.

```
In [21]: from collections import Counter

         counter = Counter('banana')
         counter
```

```
Out[21]: Counter({'a': 3, 'n': 2, 'b': 1})
```

```
In [22]: from collections import Counter

         t = (1, 1, 1, 2, 2, 3)
         counter = Counter(t)
         counter
```

```
Out[22]: Counter({1: 3, 2: 2, 3: 1})
```

A `Counter` object is like a dictionary that maps from each key to the number of times it appears. As in dictionaries, the keys have to be hashable.

Unlike dictionaries, `Counter` objects don't raise an exception if you access an element that doesn't appear. Instead, they return `0`.

```
In [23]: counter['d']
```

Out[23]: 0

We can use `Counter` objects to solve one of the exercises from Chapter 10, which asks for a function that takes two words and checks whether they are anagrams -- that is, whether the letters from one can be rearranged to spell the other.

Here's a solution using `Counter` objects.

In [24]:
```python
def is_anagram(word1, word2):
    return Counter(word1) == Counter(word2)
```

If two words are anagrams, they contain the same letters with the same counts, so their `Counter` objects are equivalent.

`Counter` provides a method called `most_common` that returns a list of value-frequency pairs, sorted from most common to least.

In [25]:
```python
counter.most_common()
```

Out[25]: [(1, 3), (2, 2), (3, 1)]

They also provide methods and operators to perform set-like operations, including addition, subtraction, union and intersection. For example, the `+` operator combines two `Counter` objects and creates a new `Counter` that contains the keys from both and the sums of the counts.

We can test it by making a `Counter` with the letters from `'bans'` and adding it to the letters from `'banana'`.

In [26]:
```python
counter2 = Counter('bans')
counter + counter2
```

Out[26]: Counter({1: 3, 2: 2, 3: 1, 'b': 1, 'a': 1, 'n': 1, 's': 1})

You'll have a chance to explore other `Counter` operations in the exercises at the end of this chapter.

defaultdict

The `collections` module also provides `defaultdict`, which is like a dictionary except that if you access a key that doesn't exist, it generates a new value automatically.

When you create a `defaultdict`, you provide a function that's used to create new values. A function that create objects is sometimes called a **factory**. The built-in functions that create lists, sets, and other types can be used as factories.

For example, here's a `defaultdict` that creates a new `list` when needed.

```
In [27]: from collections import defaultdict

         d = defaultdict(list)
         d
```

Out[27]: defaultdict(list, {})

Notice that the argument is `list`, which is a class object, not `list()`, which is a function call that creates a new list. The factory function doesn't get called unless we access a key that doesn't exist.

```
In [28]: t = d['new key']
         t
```

Out[28]: []

The new list, which we're calling `t`, is also added to the dictionary. So if we modify `t`, the change appears in `d`:

```
In [29]: t.append('new value')
         d['new key']
```

Out[29]: ['new value']

If you are making a dictionary of lists, you can often write simpler code using `defaultdict`.

In one of the exercises in Chapter 11, I made a dictionary that maps from a sorted string of letters to the list of words that can be spelled with those letters. For example, the string `'opst'` maps to the list `['opts', 'post', 'pots', 'spot', 'stop', 'tops']`. Here's the original code.

```
In [30]: def all_anagrams(filename):
             d = {}
             for line in open(filename):
                 word = line.strip().lower()
                 t = signature(word)
                 if t not in d:
                     d[t] = [word]
                 else:
                     d[t].append(word)
             return d
```

And here's a simpler version using a `defaultdict`.

```
In [31]: def all_anagrams(filename):
             d = defaultdict(list)
             for line in open(filename):
                 word = line.strip().lower()
                 t = signature(word)
```

```
        d[t].append(word)
    return d
```

In the exercises at the end of the chapter, you'll have a chance to practice using `defaultdict` objects.

In [32]:
```python
from collections import defaultdict

d = defaultdict(list)
key = ('into', 'the')
d[key].append('woods')
d[key]
```

Out[32]: `['woods']`

Conditional expressions

Conditional statements are often used to choose one of two values, like this:

In [33]:
```python
import math
x = 5
```

In [34]:
```python
if x > 0:
    y = math.log(x)
else:
    y = float('nan')
```

In [35]: `y`

Out[35]: `1.6094379124341003`

This statement checks whether `x` is positive. If so, it computes its logarithm. If not, `math.log` would raise a ValueError. To avoid stopping the program, we generate a `NaN`, which is a special floating-point value that represents "Not a Number".

We can write this statement more concisely using a **conditional expression**.

In [36]:
```python
y = math.log(x) if x > 0 else float('nan')
```

In [37]: `y`

Out[37]: `1.6094379124341003`

You can almost read this line like English: " `y` gets log- `x` if `x` is greater than 0; otherwise it gets `NaN` ".

Recursive functions can sometimes be written concisely using conditional expressions. For example, here is a version of `factorial` with a conditional *statement*.

```
In [38]: def factorial(n):
             if n == 0:
                 return 1
             else:
                 return n * factorial(n-1)
```

And here's a version with a conditional *expression*.

```
In [39]: def factorial(n):
             return 1 if n == 0 else n * factorial(n-1)
```

Another use of conditional expressions is handling optional arguments. For example, here is class definition with an `__init__` method that uses a conditional statement to check a parameter with a default value.

```
In [40]: class Kangaroo:
             def __init__(self, name, contents=None):
                 self.name = name
                 if contents is None:
                     contents = []
                 self.contents = contents
```

Here's a version that uses a conditional expression.

```
In [41]: def __init__(self, name, contents=None):
             self.name = name
             self.contents = [] if contents is None else contents
```

In general, you can replace a conditional statement with a conditional expression if both branches contain a single expression and no statements.

List comprehensions

In previous chapters, we've seen a few examples where we start with an empty list and add elements, one at a time, using the `append` method. For example, suppose we have a string that contains the title of a movie, and we want to capitalize all of the words.

```
In [42]: title = 'monty python and the holy grail'
```

We can split it into a list of strings, loop through the strings, capitalize them, and append them to a list.

```
In [43]: t = []
         for word in title.split():
             t.append(word.capitalize())

         ' '.join(t)
```

```
Out[43]:  'Monty Python And The Holy Grail'
```

We can do the same thing more concisely using a **list comprehension**:

```
In [44]:  t = [word.capitalize() for word in title.split()]

          ' '.join(t)
```

```
Out[44]:  'Monty Python And The Holy Grail'
```

The bracket operators indicate that we are constructing a new list. The expression inside the brackets specifies the elements of the list, and the `for` clause indicates what sequence we are looping through.

The syntax of a list comprehension might seem strange, because the loop variable -- `word` in this example -- appears in the expression before we get to its definition. But you get used to it.

As another example, in Chapter 9 we used this loop to read words from a file and append them to a list.

```
In [45]:  download('https://raw.githubusercontent.com/AllenDowney/ThinkPython2/master/code/wo

          Downloaded words.txt
```

```
In [46]:  word_list = []

          for line in open('words.txt'):
              word = line.strip()
              word_list.append(word)
```

```
In [47]:  len(word_list)
```

```
Out[47]:  113783
```

Here's how we can write that as a list comprehension.

```
In [48]:  word_list = [line.strip() for line in open('words.txt')]
```

```
In [49]:  len(word_list)
```

```
Out[49]:  113783
```

A list comprehension can also have an `if` clause that determines which elements are included in the list. For example, here's a `for` loop we used in Chapter 10 to make a list of only the words in `word_list` that are palindromes.

```
In [50]:  def is_palindrome(word):
              return list(reversed(word)) == list(word)
```

```
In [51]:  palindromes = []

          for word in word_list:
              if is_palindrome(word):
                  palindromes.append(word)
```

```
In [52]:  palindromes[:10]
```

```
Out[52]:  ['aa', 'aba', 'aga', 'aha', 'ala', 'alula', 'ama', 'ana', 'anna', 'ava']
```

Here's how we can do the same thing with an list comprehension.

```
In [53]:  palindromes = [word for word in word_list if is_palindrome(word)]
```

```
In [54]:  palindromes[:10]
```

```
Out[54]:  ['aa', 'aba', 'aga', 'aha', 'ala', 'alula', 'ama', 'ana', 'anna', 'ava']
```

When a list comprehension is used as an argument to a function, we can often omit the brackets. For example, suppose we want to add up $1 / 2^n$ for values of n from 0 to 9. We can use a list comprehension like this.

```
In [55]:  sum([1/2**n for n in range(10)])
```

```
Out[55]:  1.998046875
```

Or we can leave out the brackets like this.

```
In [56]:  sum(1/2**n for n in range(10))
```

```
Out[56]:  1.998046875
```

In this example, the argument is technically a **generator expression**, not a list comprehension, and it never actually makes a list. But other than that, the behavior is the same.

List comprehensions and generator expressions are concise and easy to read, at least for simple expressions. And they are usually faster than the equivalent for loops, sometimes much faster. So if you are mad at me for not mentioning them earlier, I understand.

But, in my defense, list comprehensions are harder to debug because you can't put a print statement inside the loop. I suggest you use them only if the computation is simple enough that you are likely to get it right the first time. Or consider writing and debugging a `for` loop and then converting it to a list comprehension.

`any` and `all`

Python provides a built-in function, `any`, that takes a sequence of boolean values and returns `True` if any of the values are `True`.

```
In [57]: any([False, False, True])
```

```
Out[57]: True
```

`any` is often used with generator expressions.

```
In [58]: any(letter == 't' for letter in 'monty')
```

```
Out[58]: True
```

That example isn't very useful because it does the same thing as the `in` operator. But we could use `any` to write concise solutions to some of the exercises in Chapter 7. For example, we can write `uses_none` like this.

```
In [59]: def uses_none(word, forbidden):
             """Checks whether a word avoids forbidden letters."""
             return not any(letter in forbidden for letter in word)
```

```
In [60]: uses_none('banana', 'xyz')
```

```
Out[60]: True
```

```
In [61]: uses_none('apple', 'efg')
```

```
Out[61]: False
```

This function loops through the letters in `word` and checks whether any of them are in `forbidden`. Using `any` with a generator expression is efficient because it stops immediately if it finds a `True` value, so it doesn't have to loop through the whole sequence.

Python provides another built-in function, `all`, that returns `True` if every element of the sequence is `True`. We can use it to write a concise version of `uses_all`.

```
In [62]: def uses_all(word, required):
             """Check whether a word uses all required letters."""
             return all(letter in word for letter in required)
```

```
In [63]: uses_all('banana', 'ban')
```

```
Out[63]: True
```

```
In [64]: uses_all('apple', 'api')
```

```
Out[64]: False
```

Expressions using `any` and `all` can be concise, efficient, and easy to read.

Named tuples

The `collections` module provides a function called `namedtuple` that can be used to create simple classes. For example, the `Point` object in Chapter 16 has only two attributes, `x` and `y`. Here's how we defined it.

```
In [65]: class Point:
             """Represents a point in 2-D space."""

             def __init__(self, x, y):
                 self.x = x
                 self.y = y

             def __str__(self):
                 return f'({self.x}, {self.y})'
```

That's a lot of code to convey a small amount of information. `namedtuple` provides a more concise way to define classes like this.

```
In [66]: from collections import namedtuple

         Point = namedtuple('Point', ['x', 'y'])
```

The first argument is the name of the class you want to create. The second is a list of the attributes `Point` objects should have. The result is a class object, which is why it is assigned to a capitalized variable name.

A class created with `namedtuple` provides an `__init__` method that assigns values to the attributes and a `__str__` that displays the object in a readable form. So we can create and display a `Point` object like this.

```
In [67]: p = Point(1, 2)
         p
```

```
Out[67]: Point(x=1, y=2)
```

`Point` also provides an `__eq__` method that checks whether two `Point` objects are equivalent -- that is, whether their attributes are the same.

```
In [68]: p == Point(1, 2)
```

```
Out[68]: True
```

You can access the elements of a named tuple by name or by index.

```
In [69]: p.x, p.y
```

```
Out[69]: (1, 2)
```

```
In [70]: p[0], p[1]
```

```
Out[70]: (1, 2)
```

You can also treat a named tuple as a tuple, as in this assignment.

```
In [71]: x, y = p
         x, y
```

```
Out[71]: (1, 2)
```

But `namedtuple` objects are immutable. After the attributes are initialized, they can't be changed.

```
In [72]: p[0] = 3
```

```
TypeError: 'Point' object does not support item assignment
```

```
In [73]: p.x = 3
```

```
AttributeError: can't set attribute
```

`namedtuple` provides a quick way to define simple classes. The drawback is that simple classes don't always stay simple. You might decide later that you want to add methods to a named tuple. In that case, you can define a new class that inherits from the named tuple.

```
In [74]: class Pointier(Point):
             """This class inherits from Point"""
```

Or at that point you could switch to a conventional class definition.

Packing keyword arguments

In Chapter 11, we wrote a function that packs its arguments into a tuple.

```
In [75]: def mean(*args):
             return sum(args) / len(args)
```

You can call this function with any number of arguments.

```
In [76]: mean(1, 2, 3)
```

```
Out[76]: 2.0
```

But the `*` operator doesn't pack keyword arguments. So calling this function with a keyword argument causes an error.

```
In [77]: mean(1, 2, start=3)
```

To pack keyword arguments, we can use the `**` operator:

```
In [78]: def mean(*args, **kwargs):
             print(kwargs)
             return sum(args) / len(args)
```

The keyword-packing parameter can have any name, but `kwargs` is a common choice. The result is a dictionary that maps from keywords to values.

```
In [79]: mean(1, 2, start=3)
```

```
{'start': 3}
```

Out[79]: 1.5

In this example, the value of `kwargs` is printed, but otherwise is has no effect.

But the `**` operator can also be used in an argument list to unpack a dictionary. For example, here's a version of `mean` that packs any keyword arguments it gets and then unpacks them as keyword arguments for `sum`.

```
In [80]: def mean(*args, **kwargs):
             return sum(args, **kwargs) / len(args)
```

Now if we call `mean` with `start` as a keyword argument, it gets passed along to sum, which uses it as the starting point of the summation. In the following example `start=3` adds `3` to the sum before computing the mean, so the sum is `6` and the result is `3`.

```
In [81]: mean(1, 2, start=3)
```

Out[81]: 3.0

As another example, if we have a dictionary with keys `x` and `y`, we can use it with the unpack operator to create a `Point` object.

```
In [82]: d = dict(x=1, y=2)
         Point(**d)
```

Out[82]: Point(x=1, y=2)

Without the unpack operator, `d` is treated as a single positional argument, so it gets assigned to `x`, and we get a `TypeError` because there's no second argument to assign to `y`.

```
In [83]: d = dict(x=1, y=2)
         Point(d)
```

When you are working with functions that have a large number of keyword arguments, it is often useful to create and pass around dictionaries that specify frequently used options.

```
In [84]:  def pack_and_print(**kwargs):
              print(kwargs)

          pack_and_print(a=1, b=2)
```

```
{'a': 1, 'b': 2}
```

Debugging

In previous chapters, we used `doctest` to test functions. For example, here's a function called `add` that takes two numbers and returns their sum. In includes a doctest that checks whether `2 + 2` is `4`.

```
In [85]:  def add(a, b):
              '''Add two numbers.

              >>> add(2, 2)
              4
              '''
              return a + b
```

This function takes a function object and runs its doctests.

```
In [86]:  from doctest import run_docstring_examples

          def run_doctests(func):
              run_docstring_examples(func, globals(), name=func.__name__)
```

So we can test `add` like this.

```
In [87]:  run_doctests(add)
```

There's no output, which means all tests passed.

Python provides another tool for running automated tests, called `unittest`. It is a little more complicated to use, but here's an example.

```
In [88]:  from unittest import TestCase

          class TestExample(TestCase):

              def test_add(self):
                  result = add(2, 2)
                  self.assertEqual(result, 4)
```

First we import `TestCase`, which is a class in the `unittest` module. To use it, we have to define a new class that inherits from `TestCase` and provides at least one test method. The

name of the test method must begin with `test` and should indicate which function it tests.

In this example, `test_add` tests the `add` function by calling it, saving the result, and invoking `assertEqual`, which is inherited from `TestCase`. `assertEqual` takes two arguments and checks whether they are equal.

In order to run this test method, we have to run a function in `unittest` called `main` and provide several keyword arguments. The following function shows the details -- if you are curious, you can ask a virtual assistant to explain how it works.

```
In [89]:  import unittest

          def run_unittest():
              unittest.main(argv=[''], verbosity=0, exit=False)
```

`run_unittest` does not take `TestExample` as an argument -- instead, it searches for classes that inherit from `TestCase`. Then it searches for methods that begin with `test` and runs them. This process is called **test discovery**.

Here's what happens when we call `run_unittest`.

```
In [90]:  run_unittest()
```

```
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

`unittest.main` reports the number of tests it ran and the results. In this case `OK` indicates that the tests passed.

To see what happens when a test fails, we'll add an incorrect test method to `TestExample`.

```
In [91]:  %%add_method_to TestExample

              def test_add_broken(self):
                  result = add(2, 2)
                  self.assertEqual(result, 100)
```

Here's what happens when we run the tests.

```
In [92]:  run_unittest()
```

```
======================================================================
FAIL: test_add_broken (__main__.TestExample.test_add_broken)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\colin\AppData\Local\Temp\ipykernel_26808\3833266738.py", line 3, in
test_add_broken
    self.assertEqual(result, 100)
AssertionError: 4 != 100


----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (failures=1)
```

The report includes the test method that failed and an error message showing where. The summary indicates that two tests ran and one failed.

In the exercises below, I'll suggest some prompts you can use to ask a virtual assistant for more information about `unittest` .

Glossary

factory: A function used to create objects, often passed as a parameter to a function.

conditional expression: An expression that uses a conditional to select one of two values.

list comprehension: A concise way to loop through a sequence and create a list.

generator expression: Similar to a list comprehension except that it does not create a list.

test discovery: A process used to find and run tests.

Exercises

```
In [93]:  # This cell tells Jupyter to provide detailed debugging information
          # when a runtime error occurs. Run it before working on the exercises.

          %xmode Verbose
```
Exception reporting mode: Verbose

Ask a virtual assistant

There are a few topics in this chapter you might want to learn about. Here are some question to ask an AI.

- "What are the methods and operators of Python's set class?"

- "What are the methods and operators of Python's Counter class?"

- "What is the difference between a Python list comprehension and a generator expression?"

- "When should I use Python's `namedtuple` rather than define a new class?"

- "What are some uses of packing and unpacking keyword arguments?"

- "How does `unittest` do test discovery?"

"Along with `assertEqual`, what are the most commonly used methods in `unittest.TestCase`?"

"What are the pros and cons of `doctest` and `unittest`?"

For the following exercises, consider asking an AI for help, but as always, remember to test the results.

Exercise

One of the exercises in Chapter 7 asks for a function called `uses_none` that takes a word and a string of forbidden letters, and returns `True` if the word does not use any of the letters. Here's a solution.

```python
In [94]: def uses_none(word, forbidden):
    for letter in word.lower():
        if letter in forbidden.lower():
            return False
    return True
```

Write a version of this function that uses `set` operations instead of a `for` loop. Hint: ask an AI "How do I compute the intersection of Python sets?"

You can use this outline to get started.

```python
In [95]: def uses_none(word, forbidden):
    """Checks whether a word avoid forbidden letters.

    >>> uses_none('banana', 'xyz')
    True
    >>> uses_none('apple', 'efg')
    False
    >>> uses_none('', 'abc')
    True
    """
    return False
```

```python
In [ ]:
```

```python
In [96]: from doctest import run_docstring_examples
```

```
def run_doctests(func):
    run_docstring_examples(func, globals(), name=func.__name__)
```

In [97]: `run_doctests(uses_none)`

```
**********************************************************************
File "__main__", line 4, in uses_none
Failed example:
    uses_none('banana', 'xyz')
Expected:
    True
Got:
    False
**********************************************************************
File "__main__", line 8, in uses_none
Failed example:
    uses_none('', 'abc')
Expected:
    True
Got:
    False
```

Exercise

Scrabble is a board game where the objective is to use letter tiles to spell words. For example, if we have tiles with the letters `T`, `A`, `B`, `L`, `E`, we can spell `BELT` and `LATE` using a subset of the tiles -- but we can't spell `BEET` because we don't have two `E` s.

Write a function that takes a string of letters and a word, and checks whether the letters can spell the word, taking into account how many times each letter appears.

You can use the following outline to get started.

In [98]:
```python
def can_spell(letters, word):
    """Check whether the letters can spell the word.

    >>> can_spell('table', 'belt')
    True
    >>> can_spell('table', 'late')
    True
    >>> can_spell('table', 'beet')
    False
    """
    return False
```

In [ ]:

In [99]: `run_doctests(can_spell)`

```
**********************************************************************
File "__main__", line 4, in can_spell
Failed example:
    can_spell('table', 'belt')
Expected:
    True
Got:
    False
**********************************************************************
File "__main__", line 6, in can_spell
Failed example:
    can_spell('table', 'late')
Expected:
    True
Got:
    False
```

Exercise

In one of the exercises from Chapter 17, my solution to `has_straightflush` uses the following method, which partitions a `PokerHand` into a list of four hands, where each hand contains cards of the same suit.

In [100…
```python
def partition(self):
    """Make a list of four hands, each containing only one suit."""
    hands = []
    for i in range(4):
        hands.append(PokerHand())

    for card in self.cards:
        hands[card.suit].add_card(card)

    return hands
```

Write a simplified version of this function using a `defaultdict`.

Here's an outline of the `PokerHand` class and the `partition_suits` function you can use to get started.

In [101…
```python
class PokerHand(Hand):

    def partition(self):
        return {}
```

In [ ]:

To test your code, we'll make a deck and shuffle it.

In [102…
```python
cards = Deck.make_cards()
deck = Deck(cards)
deck.shuffle()
```

Then create a `PokerHand` and add seven cards to it.

```
In [103…    random_hand = PokerHand('random')

            for i in range(7):
                card = deck.pop_card()
                random_hand.add_card(card)

            print(random_hand)
```

```
6 of Diamonds
4 of Diamonds
4 of Spades
King of Spades
Ace of Clubs
8 of Hearts
5 of Hearts
```

If you invoke `partition` and print the results, each hand should contain cards of one suit only.

```
In [104…    hand_dict = random_hand.partition()

            for hand in hand_dict.values():
                print(hand)
                print()
```

Exercise

Here's the function from Chapter 11 that computes Fibonacci numbers.

```
In [105…    def fibonacci(n):
                if n == 0:
                    return 0

                if n == 1:
                    return 1

                return fibonacci(n-1) + fibonacci(n-2)
```

Write a version of this function with a single return statement that use two conditional expressions, one nested inside the other.

Begin work done by Colin Howard

```
In [106…    def fibonacci(n):
                return 0 if n==0 else 1 if n==1 else fibonacci(n-1)+fibonacci(n-2)
```

End work done by Colin Howard (*testing below*)

```
In [107…    fibonacci(10)      # should be 55
```

```
Out[107...   55
```

```
In [108...   fibonacci(20)       # should be 6765
```

```
Out[108...   6765
```

Exercise

The following is a function that computes the binomial coefficient recursively.

```
In [109...   def binomial_coeff(n, k):
                 """Compute the binomial coefficient "n choose k".

                 n: number of trials
                 k: number of successes

                 returns: int
                 """
                 if k == 0:
                     return 1

                 if n == 0:
                     return 0

                 return binomial_coeff(n-1, k) + binomial_coeff(n-1, k-1)
```

Rewrite the body of the function using nested conditional expressions.

This function is not very efficient because it ends up computing the same values over and over. Make it more efficient by memoizing it, as described in Chapter 10.

Begin work done by Colin Howard

```
In [115...   known = {}

             def binomial_coeff(n, k):
                 """Compute the binomial coefficient "n choose k".

                 n: number of trials
                 k: number of successes

                 returns: int
                 """
                 if (n,k) in known:
                     return known[n,k]

                 result = 1 if k==0 else 0 if n==0 else binomial_coeff(n-1, k) + binomial_coeff(
                 known[n,k] = result
                 return result
```

End work done by Colin Howard (*testing below*)

```
In [116…   binomial_coeff(10, 4)    # should be 210
```

```
Out[116…   210
```

Exercise

Here's the `__str__` method from the `Deck` class in Chapter 17.

```
In [111…   %%add_method_to Deck

           def __str__(self):
               res = []
               for card in self.cards:
                   res.append(str(card))
               return '\n'.join(res)
```

Write a more concise version of this method with a list comprehension or generator
expression.

Begin work done by Colin Howard

```
In [117…   %%add_method_to Deck

           def __str__(self):
               res = [str(card) for card in self.cards]
               return '\n'.join(res)
```

End work done by Colin Howard (*testing below*)

You can use this example to test your solution.

```
In [118…   cards = Deck.make_cards()
           deck = Deck(cards)
           print(deck)
```

```
2 of Clubs
3 of Clubs
4 of Clubs
5 of Clubs
6 of Clubs
7 of Clubs
8 of Clubs
9 of Clubs
10 of Clubs
Jack of Clubs
Queen of Clubs
King of Clubs
Ace of Clubs
2 of Diamonds
3 of Diamonds
4 of Diamonds
5 of Diamonds
6 of Diamonds
7 of Diamonds
8 of Diamonds
9 of Diamonds
10 of Diamonds
Jack of Diamonds
Queen of Diamonds
King of Diamonds
Ace of Diamonds
2 of Hearts
3 of Hearts
4 of Hearts
5 of Hearts
6 of Hearts
7 of Hearts
8 of Hearts
9 of Hearts
10 of Hearts
Jack of Hearts
Queen of Hearts
King of Hearts
Ace of Hearts
2 of Spades
3 of Spades
4 of Spades
5 of Spades
6 of Spades
7 of Spades
8 of Spades
9 of Spades
10 of Spades
Jack of Spades
Queen of Spades
King of Spades
Ace of Spades
```

In [ ]:

Think Python: 3rd Edition