Advanced Python 3 Training

with examples and
hands-on exercises

WEBUCATOR

Version: 1.2.3

The Authors

Nat Dunn

Nat Dunn is the founder of Webucator (www.webucator.com), a company that has provided training for tens of thousands of students from thousands of organizations. Nat started the company in 2003 to combine his passion for technical training with his business expertise, and to help companies benefit from both. His previous experience was in sales, business and technical training, and management. Nat has an MBA from Harvard Business School and a BA in International Relations from Pomona College.

Follow Nat on Twitter at @natdunn and Webucator at @webucator.

Stephen Withrow (Editor)

Stephen has over 30 years of experience in training, development, and consulting in a variety of technology areas including Python, Java, C, C++, XML, JavaScript, Tomcat, JBoss, Oracle, and DB2. His background includes design and implementation of business solutions on client/server, Web, and enterprise platforms. Stephen has a degree in Computer Science and Physics from Florida State University.

Class Files

Download the class files used in this manual at https://static.webucator.com/media/public/materials/classfiles/PYT238-1.2.3.zip.

Errata

Corrections to errors in the manual can be found at https://www.webucator.com/books/errata/.

Acknowledgments

A huge thanks to the phenomenal trainers who have taught Python using variations of this material for Webucator over the years: Stephen Withrow, Roger Sakowski, Mark Copley, Jared Dunn, and others.

And one final thanks to all the hard work the Python team does. You can join Webucator in supporting them at `https://www.python.org/psf/sponsorship/sponsors/`.

Table of Contents

LESSON 1
Advanced Python Concepts

Topics Covered

☑ Lambda functions.

☑ Advanced list comprehensions.

☑ The `collections` module.

☑ Mapping and filtering.

☑ Sorting sequences.

☑ Unpacking sequences in function calls.

☑ Modules and packages.

Introduction

In this lesson, you will learn about some Python functionality and techniques that are commonly used but require a solid foundation in Python to understand.

1.1. Lambda Functions

Lambda functions are anonymous functions that are generally used to complete a small task, after which they are no longer needed. The syntax for creating a lambda function is:

```
lambda arguments: expression
```

Lambda functions are almost always used within other functions, but for demonstration purposes, we could assign a lambda function to a variable, like this:

```
f = lambda n: n**2
```

We could then call `f` like this:

```
f(5) # Returns 25
f(2) # Returns 4
```

Try it at the Python terminal:

```
>>> f = lambda n: n**2
>>> f(5)
25
>>> f(2)
4
```

We will revisit lambda functions throughout this lesson.

1.2. Advanced List Comprehensions

❖ 1.2.1. Quick Review of Basic List Comprehensions

Before we get into advanced list comprehensions, let's do a quick review. The basic syntax for a list comprehension is:

```
my_list = [f(x) for x in iterable if condition]
```

In the preceding code `f(x)` could be any of the following:

1. Just a variable name (e.g., `x`).
2. An operation (e.g., `x**2`).
3. A function call (e.g., `len(x)` or `square(x)`).

Here is an example from earlier, in which we create a list by filtering another list:

Demo 1.1: advanced-python-concepts/Demos/sublist_from_list.py

```
1.    def main():
2.        words = ['Woodstock', 'Gary', 'Tucker', 'Gopher', 'Spike', 'Ed',
3.                 'Faline', 'Willy', 'Rex', 'Rhino', 'Roo', 'Littlefoot',
4.                 'Bagheera', 'Remy', 'Pongo', 'Kaa', 'Rudolph', 'Banzai',
5.                 'Courage', 'Nemo', 'Nala', 'Alvin', 'Sebastian', 'Iago']
6.        three_letter_words = [w for w in words if len(w) == 3]
7.        print(three_letter_words)
8.
9.    main()
```

This will return:

```
['Rex', 'Roo', 'Kaa']
```

And here is a new example, in which we map all the elements in one list to another using a function:

Demo 1.2: advanced-python-concepts/Demos/list_comp_mapping.py

```
1.    def get_inits(name):
2.        # Create list from first letter of each name part
3.        inits = [name_part[0] for name_part in name.split()]
4.        # Join inits list on "." and append "." to end
5.        return '.'.join(inits) + '.'
6.
7.    def main():
8.        people = ['George Washington', 'John Adams',
9.                  'Thomas Jefferson', 'John Quincy Adams']
10.
11.       # Create list by mapping person elements to get_inits()
12.       inits = [get_inits(person) for person in people]
13.       print(inits)
14.
15.   main()
```

This will return:

```
['G.W.', 'J.A.', 'T.J.', 'J.Q.A.']
```

Now, on to the more advanced uses of list comprehension.

❖ 1.2.2. Multiple for Loops

Assume that you need to create a list of tuples showing the possible *permutations* of rolling two six-sided dice. When dealing with permutations, order matters, so (1, 2) and (2, 1) are not the same. First, let's look at how we would do this without a list comprehension, using a nested `for` loop:

Demo 1.3: advanced-python-concepts/Demos/dice_rolls.py

```
1.   def main():
2.       dice_rolls = []
3.       for a in range(1, 7):
4.           for b in range(1, 7):
5.               roll = (a, b)
6.               dice_rolls.append(roll)
7.
8.       print(dice_rolls)
9.
10.  main()
```


This will return:

```
[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6),
(2, 1), (2, 2), (2, 3), (2, 4), (2, 5), (2, 6),
(3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (3, 6),
(4, 1), (4, 2), (4, 3), (4, 4), (4, 5), (4, 6),
(5, 1), (5, 2), (5, 3), (5, 4), (5, 5), (5, 6),
(6, 1), (6, 2), (6, 3), (6, 4), (6, 5), (6, 6)]
```

List comprehensions can include multiple `for` loops with each subsequent loop nested within the previous loop. This provides an easy way to create something similar to a two-dimensional array or a matrix:

Demo 1.4: advanced-python-concepts/Demos/dice_rolls_list_comp.py

```
1.   def main():
2.       dice_rolls = [
3.           (a, b)
4.           for a in range(1, 7)
5.           for b in range(1, 7)
6.       ]
7.
8.       print(dice_rolls)
9.
10.  main()
```

This code will create the same list of tuples containing all the possible permutations of two dice rolls.

Notice that the list of permutations contains what game players would consider duplicates. For example, `(1, 2)` and `(2, 1)` are considered the same in dice. We can remove these pseudo-duplicates by starting the second `for` loop with the current value of `a` in the first `for` loop. Let's do this first without a list comprehension:

Demo 1.5: advanced-python-concepts/Demos/dice_combos.py

```
1.   def main():
2.       dice_rolls = []
3.       for a in range(1, 7):
4.           for b in range(a, 7):
5.               roll = (a, b)
6.               dice_rolls.append(roll)
7.
8.       print(dice_rolls)
9.
10.  main()
```

The first time through the outer loop, the inner loop from 1 to 7 (not including 7), the second time through, it will loop from 2 to 7, then 3 to 7, and so on…

The `dice_rolls` list will now contain the different possible rolls (from a dice rolling point of view):

```
[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6),
(2, 2), (2, 3), (2, 4), (2, 5), (2, 6),
(3, 3), (3, 4), (3, 5), (3, 6),
(4, 4), (4, 5), (4, 6),
(5, 5), (5, 6),
(6, 6)]
```

Where we previously showed *permutations*, in which order matters, we are now showing *combinations*, in which order does not matter. The following two tuples represent different permutations, but the same combination: `(1, 2)` and `(2, 1)`.

Now, let's see how we can do the same thing with a list comprehension:

Demo 1.6: advanced-python-concepts/Demos/dice_combos_list_comp.py

```
1.  def main():
2.      dice_rolls = [
3.          (a, b)
4.          for a in range(1, 7)
5.          for b in range(a, 7)
6.      ]
7.
8.      print(dice_rolls)
9.
10. main()
```

This code will create the same list of tuples containing all the possible combinations of two dice rolls.

Exercise 1: Rolling Five Dice

10 to 15 minutes

There is no limit to the number of `for` loops in a list comprehension, so we can use this same technique to get the possibilities for more than two dice.

1. Create a new file in `advanced-python-concepts/Exercises` named `list_comprehensions.py`.
2. Write two separate list comprehensions:
    A. The first should create five-item tuples for all unique **permutations** from rolling five identical six-sided dice. Remember, when looking for permutations, order matters.
    B. The second should create five-item tuples for all unique **combinations** from rolling five identical six-sided dice. Remember, when looking for combinations, order doesn't matter.
3. Print the length of each list.

Solution: advanced-python-concepts/Solutions/list_comprehensions.py

```
1.   # Get unique permutations:
2.   dice_rolls_p = [(a, b, c, d, e)
3.                   for a in range(1, 7)
4.                   for b in range(1, 7)
5.                   for c in range(1, 7)
6.                   for d in range(1, 7)
7.                   for e in range(1, 7)]
8.
9.   print('Number of permutations:', len(dice_rolls_p))
10.
11.  # Get unique combinations:
12.  dice_rolls_c = [(a, b, c, d, e)
13.                  for a in range(1,7 )
14.                  for b in range(a, 7)
15.                  for c in range(b, 7)
16.                  for d in range(c, 7)
17.                  for e in range(d, 7)]
18.
19.  print('Number of combinations:', len(dice_rolls_c))
```

This file will output:


```
Number of permutations: 7776
Number of combinations: 252
```

1.3. Collections Module

The `collections` module includes specialized containers (objects that hold data) that provide more specific functionality than Python's built-in containers (`list`, `tuple`, `dict`, and `set`). Some of the more useful containers are *named tuples* (created with the `namedtuple()` function), `defaultdict`, and `Counter`.

❖ 1.3.1. Named Tuples

Imagine you are creating a game in which you need to set and get the position of a target. You could do this with a regular tuple like this:

```
# Set target position:
target_pos = (100, 200)

# Get x value of target position
target_pos[0] # 100
```

But someone reading your code might not understand what `target_pos[0]` refers to.

A named tuple allows you to reference `target_pos.x`, which is more meaningful and helpful. Here is a simplified signature for creating `namedtuple` objects:

```
namedtuple(typename, field_names)
```

1. `typename` – The value passed in for `typename` will be the name of a new tuple subclass. It is standard for the name of the new subclass to begin with a capital letter. We have not yet covered classes and subclasses yet. For now, it is enough to know that the new tuple subclass created by `namedtuple()` will inherit all the properties of a tuple, and also make it possible to refer to elements of the tuple by name.
2. `field_names` – The value for `field_names` can either be a whitespace-delimited string (e.g., `'x y'`), a comma-delimited string (e.g., `'x, y'`), or a sequence of strings (e.g., `['x', 'y']`).

Demo 1.7: advanced-python-concepts/Demos/namedtuple.py

```
1.  from collections import namedtuple
2.
3.  Point = namedtuple('Point', 'x, y')
4.
5.  # Set target position:
6.  target_pos = Point(100, 200)
7.
8.  # Get x value of target position
9.  print(target_pos.x)
```

As the preceding code shows, the `namedtuple()` function allows you to give a name to the elements at different positions in a tuple and then refer to them by that name.

❖ 1.3.2. Default Dictionaries (defaultdict)

With regular dictionaries, trying to modify a key that doesn't exist will cause an exception. For example, the following code will result in a `KeyError`:

```
foo = {}
foo['bar'] += 1
```

A `defaultdict` is like a regular dictionary except that, when you look up a key that doesn't exist, it creates the key and assigns it the value returned by a function specified when creating it.

To illustrate how a `defaultdict` can be useful, let's see how we would create a *regular dictionary* that shows the number of different ways each number (2 through 12) can be rolled when rolling two dice, like this:

```
{
    2: 1,
    3: 2,
    4: 3,
    5: 4,
    6: 5,
    7: 6,
    8: 5,
    9: 4,
    10: 3,
    11: 2,
    12: 1
}
```


- There is only one way to roll a 2: `(1, 1)`.
- There are two ways to roll a 3: `(1, 2)` and `(2, 1)`.
- There are five ways to roll a 6: `(1, 5)`, `(2, 4)`, `(3, 3)`, `(4, 2)`, and `(5, 1)`.

1. First, create the list of possibilities as we did earlier:

```
dice_rolls = [
    (a, b)
    for a in range(1, 7)
    for b in range(1, 7)
]
```

2. Next, create an empty dictionary, `roll_counts`, and then loop through the `dice_rolls` list checking for the existence of a key that is the sum of the dice roll. For example, on the first iteration, we find `(1, 1)`, which when added together, gives us `2`. Since `roll_counts` does not have a key `2`, we need to add that key and set its value to `1`. The same is true for when we find `(1, 2)`, which adds up to `3`. But later when we find `(2, 1)`, which also adds up to `3`, we don't need to recreate the key. Instead, we increment the existing key's value by `1`. The code looks like this:

```
roll_counts = {}
for roll in dice_rolls:
    if sum(roll) in roll_counts:
        roll_counts[sum(roll)] += 1
    else:
        roll_counts[sum(roll)] = 1
```

This method works fine and gives us the following `roll_counts` dictionary that we looked at earlier:

```
{
    2: 1,
    3: 2,
    4: 3,
    5: 4,
    6: 5,
    7: 6,
    8: 5,
    9: 4,
    10: 3,
    11: 2,
    12: 1
}
```


An alternative to using conditionals to make sure the key exists is to just go ahead and try to increment the value of each potential key we find and then, if we get a `KeyError`, assign `1` for that key, like this:

```
roll_counts = {}
for roll in dice_rolls:
    try:
        roll_counts[sum(roll)] += 1
    except KeyError:
        roll_counts[sum(roll)] = 1
```

This also works and produced the same dictionary.

But with a `defaultdict`, we don't need the `if-else` block or the `try-except` block. The code looks like this:

```
from collections import defaultdict

roll_counts = defaultdict(int)
for roll in dice_rolls:
    roll_counts[sum(roll)] += 1
```

The result is a `defaultdict` object that can be treated just like a normal dictionary:

```
defaultdict(<class 'int'>, {
    2: 1,
    3: 2,
    4: 3,
    5: 4,
    6: 5,
    7: 6,
    8: 5,
    9: 4,
    10: 3,
    11: 2,
    12: 1
})
```


Here are the three methods again:

Demo 1.8: advanced-python-concepts/Demos/dict_if_else.py

```
     -------Lines 1 through 4 Omitted-------
5.   roll_counts = {}
6.   for roll in dice_rolls:
7.       if sum(roll) in roll_counts:
8.           roll_counts[sum(roll)] += 1
9.       else:
10.          roll_counts[sum(roll)] = 1
```

Demo 1.9: advanced-python-concepts/Demos/dict_try_except.py

```
     -------Lines 1 through 4 Omitted-------
5.   roll_counts = {}
6.   for roll in dice_rolls:
7.       try:
8.           roll_counts[sum(roll)] += 1
9.       except KeyError:
10.          roll_counts[sum(roll)] = 1
```

Demo 1.10: advanced-python-concepts/Demos/defaultdict.py

```
1.   from collections import defaultdict
     -------Lines 2 through 6 Omitted-------
7.   roll_counts = defaultdict(int)
8.   for roll in dice_rolls:
9.       roll_counts[sum(roll)] += 1
```

Notice in the preceding code that we passed `int` to `defaultdict()`:


```
roll_counts = defaultdict(int)
```

Remember, when you try to look up a key that doesn't exist in a `defaultdict`, it creates the key and assigns it the value returned by a function you specified when creating it. In this case, that function is `int()`.

When passing the function to `defaultdict()`, you do not include parentheses, because you are not calling the function at the time you pass it to `defaultdict()`. Rather, you are specifying that you want to use this function to give you default values for new keys. By passing `int`, we are stating that we want new keys to have a default value of whatever `int()` returns when no argument is passed to it. That value is `0`:

```
>>> int()
0
```

You can create default dictionaries with any number of functions, both built-in and user-defined:

defaultdict with Built-in Functions

```
a = defaultdict(list) # Default key value will be []
b = defaultdict(str) # Default key value will be ''
```

defaultdict with lambda Function

```
c = defaultdict(lambda: 5) # Default key value will be 5
c['a'] += 1 # c['a'] will contain 6
```


defaultdict with User-defined Function

```
def foo():
    return 'bar'

d = defaultdict(foo) # Default key value will be 'bar'
d['a'] = d['a'].upper() # d['a'] will contain 'BAR'
```

Exercise 2: Creating a defaultdict

15 to 25 minutes

In this exercise, you will organize the 1927 New York Yankees by position by creating a default dictionary that looks like this:

```
defaultdict(<class 'list'>,
{
    'OF': ['Earle Combs', 'Cedric Durst', 'Bob Meusel',
           'Ben Paschal', 'Babe Ruth'],
    'C': ['Benny Bengough', 'Pat Collins', 'Johnny Grabowski'],
    '2B': ['Tony Lazzeri', 'Ray Morehart'],
    'SS': ['Mark Koenig'],
    '3B': ['Joe Dugan', 'Mike Gazella', 'Julie Wera'],
    'P': ['Walter Beall', 'Joe Giard', 'Waite Hoyt',
          'Wilcy Moore', 'Herb Pennock', 'George Pipgras',
          'Dutch Ruether', 'Bob Shawkey', 'Urban Shocker',
          'Myles Thomas'],
    '1B': ['Lou Gehrig']
})
```


You will start with this list of dictionaries:

```
yankees_1927 = [
    {'position': 'P', 'name': 'Walter Beall'},
    {'position': 'C', 'name': 'Benny Bengough'},
    {'position': 'C', 'name': 'Pat Collins'},
    {'position': 'OF', 'name': 'Earle Combs'},
    {'position': '3B', 'name': 'Joe Dugan'},
    {'position': 'OF', 'name': 'Cedric Durst'},
    {'position': '3B', 'name': 'Mike Gazella'},
    {'position': '1B', 'name': 'Lou Gehrig'},
    {'position': 'P', 'name': 'Joe Giard'},
    {'position': 'C', 'name': 'Johnny Grabowski'},
    {'position': 'P', 'name': 'Waite Hoyt'},
    {'position': 'SS', 'name': 'Mark Koenig'},
    {'position': '2B', 'name': 'Tony Lazzeri'},
    {'position': 'OF', 'name': 'Bob Meusel'},
    {'position': 'P', 'name': 'Wilcy Moore'},
    {'position': '2B', 'name': 'Ray Morehart'},
    {'position': 'OF', 'name': 'Ben Paschal'},
    {'position': 'P', 'name': 'Herb Pennock'},
    {'position': 'P', 'name': 'George Pipgras'},
    {'position': 'P', 'name': 'Dutch Ruether'},
    {'position': 'OF', 'name': 'Babe Ruth'},
    {'position': 'P', 'name': 'Bob Shawkey'},
    {'position': 'P', 'name': 'Urban Shocker'},
    {'position': 'P', 'name': 'Myles Thomas'},
    {'position': '3B', 'name': 'Julie Wera'}
]
```


1. Open `advanced-python-concepts/Exercises/defaultdict.py` in your editor.
2. Write code so that the script creates the `defaultdict` above from the given list.
3. Output the pitchers stored in your new `defaultdict`.

Solution: advanced-python-concepts/Solutions/defaultdict.py

```
1.  from collections import defaultdict
2.
3.  yankees_1927 = [
4.      {'position': 'P', 'name': 'Walter Beall'},
5.      {'position': 'C', 'name': 'Benny Bengough'},
6.      {'position': 'C', 'name': 'Pat Collins'},
7.      {'position': 'OF', 'name': 'Earle Combs'},
8.      {'position': '3B', 'name': 'Joe Dugan'},
9.      {'position': 'OF', 'name': 'Cedric Durst'},
10.     {'position': '3B', 'name': 'Mike Gazella'},
11.     {'position': '1B', 'name': 'Lou Gehrig'},
12.     {'position': 'P', 'name': 'Joe Giard'},
13.     {'position': 'C', 'name': 'Johnny Grabowski'},
14.     {'position': 'P', 'name': 'Waite Hoyt'},
15.     {'position': 'SS', 'name': 'Mark Koenig'},
16.     {'position': '2B', 'name': 'Tony Lazzeri'},
17.     {'position': 'OF', 'name': 'Bob Meusel'},
18.     {'position': 'P', 'name': 'Wilcy Moore'},
19.     {'position': '2B', 'name': 'Ray Morehart'},
20.     {'position': 'OF', 'name': 'Ben Paschal'},
21.     {'position': 'P', 'name': 'Herb Pennock'},
22.     {'position': 'P', 'name': 'George Pipgras'},
23.     {'position': 'P', 'name': 'Dutch Ruether'},
24.     {'position': 'OF', 'name': 'Babe Ruth'},
25.     {'position': 'P', 'name': 'Bob Shawkey'},
26.     {'position': 'P', 'name': 'Urban Shocker'},
27.     {'position': 'P', 'name': 'Myles Thomas'},
28.     {'position': '3B', 'name': 'Julie Wera'}
29. ]
30.
31. # Each value will be a list of players, so we pass list to defaultdict
32. positions = defaultdict(list)
33.
34. # Loop through list of yankees appending player names to their position keys
35. for player in yankees_1927:
36.     positions[player['position']].append(player['name'])
37.
38. print(positions['P'])
```


This will output the list of pitchers:

```
[
'Walter Beall',
'Joe Giard',
'Waite Hoyt',
'Wilcy Moore',
'Herb Pennock',
'George Pipgras',
'Dutch Ruether',
'Bob Shawkey',
'Urban Shocker',
'Myles Thomas'
]
```

Add the following line of code to the `for` loop to watch as the players get added:

```
print(player['position'], positions[player['position']])
```

The beginning of the output will look like this:


```
…/advanced-python-concepts/Solutions> python defaultdict.py
P ['Walter Beall']
C ['Benny Bengough']
C ['Benny Bengough', 'Pat Collins']
OF ['Earle Combs']
3B ['Joe Dugan']
OF ['Earle Combs', 'Cedric Durst']
3B ['Joe Dugan', 'Mike Gazella']
1B ['Lou Gehrig']
P ['Walter Beall', 'Joe Giard']
C ['Benny Bengough', 'Pat Collins', 'Johnny Grabowski']
…
```

1.4. Counters

Consider again the `defaultdict` object we created to get the number of different ways each number could be rolled when rolling two dice. This type of task is very common. You might have a collection of plants and want to get a count of the number of each species or the number of plants by color. The

objects that hold these counts are called *counters*, and the `collections` module includes a special `Counter()` class for creating them.

Although there are different ways of creating counters, they are most often created with an iterable, like this:

```
from collections import Counter
c = Counter(['green', 'blue', 'blue', 'red', 'yellow', 'green', 'blue'])
```

This will create the following counter:

```
Counter({
    'blue': 3,
    'green': 2,
    'red': 1,
    'yellow': 1
})
```

To create a counter from the `dice_rolls` list we used earlier, we need to first create a list of sums from it, like this:


```
roll_sums = [sum(roll) for roll in dice_rolls]
```

`roll_sums` will contain the following list:

```
[
    2, 3, 4, 5, 6, 7,
    3, 4, 5, 6, 7, 8,
    4, 5, 6, 7, 8, 9,
    5, 6, 7, 8, 9, 10,
    6, 7, 8, 9, 10, 11,
    7, 8, 9, 10, 11, 12
]
```

We then create the counter like this:

```
c = Counter(roll_sums)
```

That creates a counter that is very similar to the `defaultdict` we saw earlier:

```
Counter({
    7: 6,
    6: 5,
    8: 5,
    5: 4,
    9: 4,
    4: 3,
    10: 3,
    3: 2,
    11: 2,
    2: 1,
    12: 1
})
```

The code in the following file creates and outputs the colors and dice rolls counters:

Demo 1.11: advanced-python-concepts/Demos/counter.py


```
1.   from collections import Counter
2.   c = Counter(['green','blue','blue','red','yellow','green','blue'])
3.   print('Colors Counter:', c, sep='\n', end='\n\n')
4.
5.   dice_rolls = [(a,b)
6.                 for a in range(1,7)
7.                 for b in range(1,7)]
8.
9.   roll_sums = [sum(roll) for roll in dice_rolls]
10.  c = Counter(roll_sums)
11.  print('Dice Roll Counter:', c, sep='\n')
```

Run this file. It will output:

```
Colors Counter:
Counter({'blue': 3, 'green': 2, 'red': 1, 'yellow': 1})

Dice Roll Counter:
Counter({7: 6, 6: 5, 8: 5, 5: 4, 9: 4, 4: 3, 10: 3, 3: 2, 11: 2, 2: 1, 12: 1})
```

Updating Counters

`Counter` is a subclass of `dict`. We will learn more about subclasses later, but for now all you need to understand is that a subclass generally has access to all of its superclass's methods and data. So, `Counter` supports all the standard `dict` instance methods. The `update()` method behaves differently though. In standard `dict` objects, `update()` replaces key values with those of the passed-in dictionary:

Updating with a Dictionary

```
grades = {'English':97, 'Math':93, 'Art':74, 'Music':86}
grades.update({'Math':97, 'Gym':93})
```

The `grades` dictionary will now contain:

```
{
    'English': 97,
    'Math': 97, # 97 replaces 93
    'Art': 74,
    'Music': 86,
    'Gym': 93 # Key is added with value of 93
}
```


In `Counter` objects, `update()` adds the values of a passed-in iterable or another `Counter` object to its own values:

Updating a Counter with a List

```
>>> c = Counter(['green', 'blue', 'blue', 'red', 'yellow', 'green', 'blue'])
>>> c # Before update:
Counter({'blue': 3, 'green': 2, 'red': 1, 'yellow': 1})
>>> c.update(['red', 'yellow', 'yellow', 'purple'])
>>> c # After update:
Counter({'blue': 3, 'yellow': 3, 'green': 2, 'red': 2, 'purple': 1})
```

Updating a Counter with a Counter

```
>>> c = Counter(['green', 'blue', 'blue', 'red', 'yellow', 'green', 'blue'])
>>> d = Counter(['green', 'violet'])
>>> c.update(d)
>>> c
Counter({'green': 3, 'blue': 3, 'red': 1, 'yellow': 1, 'violet': 1})
```

Counters also have a corresponding `subtract()` method. It works just like `update()` but subtracts rather than adds the passed-in iterable counts:

Subtracting with a Counter

```
>>> c = Counter(['green', 'blue', 'blue', 'red', 'yellow', 'green', 'blue'])
>>> c # Before subtraction:
Counter({'blue': 3, 'green': 2, 'red': 1, 'yellow': 1})
>>> c.subtract(['red', 'yellow', 'yellow', 'purple'])
>>> c # After subtraction:
Counter({'blue': 3, 'green': 2, 'red': 0, 'yellow': -1, 'purple': -1})
```

Notice that the value for the 'yellow' and 'purple' keys are negative, which is a little odd. We will learn how to create a non-negative counter in a later lesson. (see page 194)

The most_common([n]) Method

Counters include a `most_common([n])` method that returns the `n` most common elements and their counts, sorted from most to least common. If `n` is not passed in, all elements are returned.

```
>>> c = Counter(['green', 'blue', 'blue', 'red', 'yellow', 'green', 'blue'])
>>> c.most_common()
[('blue', 3), ('green', 2), ('red', 1), ('yellow', 1)]
>>> c.most_common(2)
[('blue', 3), ('green', 2)]
```

Exercise 3: Creating a Counter

10 to 15 minutes

In this exercise, you will create a counter that holds the most common words used and the number of times they show up in the *U.S. Declaration of Independence*.

1. Create a new Python script in `advanced-python-concepts/Exercises` named `counter.py`.
2. Write code that:
    A. Reads the `Declaration_of_Independence.txt` file in the same folder.
    B. Creates a list of all the words that have at least six characters.
        - Use `split()` to split the text into words. This will split the text on whitespace. This isn't quite correct as it doesn't account for punctuation, but for now, it's good enough.
        - Use `upper()` to convert the words to uppercase.
    C. Creates a `Counter` from the word list.
    D. Outputs the most common ten words and their counts. The result should look like this:

```
[
    ('PEOPLE', 13),
    ('STATES', 7),
    ('SHOULD', 5),
    ('INDEPENDENT', 5),
    ('AGAINST', 5),
    ('GOVERNMENT,', 4),
    ('ASSENT', 4),
    ('OTHERS', 4),
    ('POLITICAL', 3),
    ('POWERS', 3)
]
```

Solution: advanced-python-concepts/Solutions/counter.py

```
1.  from collections import Counter
2.
3.  with open('Declaration_of_Independence.txt') as f:
4.      doi = f.read()
5.
6.  word_list = [word for word in doi.upper().split() if len(word) > 5]
7.
8.  c = Counter(word_list)
9.  print(c.most_common(10))
```

1.5. Mapping and Filtering

❖ 1.5.1. map(function, iterable, …)

The built-in `map()` function is used to sequentially pass all the values of an iterable (or multiple iterables) to a function and return an iterator containing the returned values. It can be used as an alternative to list comprehensions. First, consider the following code sample that does not use `map()`:

Demo 1.12: advanced-python-concepts/Demos/without_map.py

```
1.  def multiply(x, y):
2.      return x * y
3.
4.  def main():
5.      nums1 = range(0, 10)
6.      nums2 = range(10, 0, -1)
7.
8.      multiples = []
9.      for i in range(len(nums1)):
10.         multiple = multiply(nums1[i], nums2[i])
11.         multiples.append(multiple)
12.
13.     for multiple in multiples:
14.         print(multiple)
15.
16. main()
```

This code creates an iterator (a list) by multiplying `0` by `10`, `1` by `9`, `2` by `8`,… and `9` by `1`. It then loops through the iterator printing each result. It will output:

```
0
9
16
21
24
25
24
21
16
9
```

The following code sample does the same thing using `map()`:

Demo 1.13: advanced-python-concepts/Demos/with_map.py

```
1.   def multiply(x, y):
2.       return x * y
3.
4.   def main():
5.       nums1 = range(0, 10)
6.       nums2 = range(10, 0, -1)
7.
8.       multiples = map(multiply, nums1, nums2)
9.
10.      for multiple in multiples:
11.          print(multiple)
12.
13.  main()
```


We could also include the `map()` function right in the `for` loop:

```
for multiple in map(multiply, nums1, nums2):
    print(multiple)
```

Note that you can accomplish the same thing with a list comprehension:

```
multiples = [multiply(nums1[i], nums2[i]) for i in range(len(nums1))]
```

One possible advantage of using `map()` in combination with multiple sequences is that it will not error if the sequences are different lengths. It will stop mapping when it reaches the end of the shortest sequence. In some cases, this might also be a disadvantage as it might hide a bug in the code. Also, this "feature" can be reproduced easily enough using the `min()` function:

```
[multiply(nums1[i], nums2[i]) for i in range(min(len(nums1), len(nums2)))]
```

❖ 1.5.2. filter(function, iterable)

The built-in `filter()` function is used to sequentially pass all the values of a single iterable to a function and return an iterator containing the values for which the function returns `True`. As with `map()`, `filter()` can be used as an alternative to list comprehensions. First, consider the following code sample that does not use `filter()`:

Demo 1.14: advanced-python-concepts/Demos/without_filter.py

```
1.  def is_odd(num):
2.      return num % 2
3.
4.  def main():
5.      nums = range(0, 10)
6.
7.      odd_nums = []
8.      for num in nums:
9.          if is_odd(num):
10.             odd_nums.append(num)
11.
12.     for num in odd_nums:
13.         print(num)
14.
15. main()
```


This code passes a range of numbers one by one to the `is_odd()` function to create an iterator (a list) of odd numbers. It then loops through the iterator printing each result. It will output:

```
1
3
5
7
9
```

The following code sample does the same thing using `filter()`:

Demo 1.15: advanced-python-concepts/Demos/with_filter.py

```
1.    def is_odd(num):
2.        return num % 2
3.
4.    def main():
5.        nums = range(0, 10)
6.
7.        odd_nums = filter(is_odd, nums)
8.
9.        for num in odd_nums:
10.           print(num)
11.
12.   main()
```


As with `map()`, we can include the `filter()` function right in the `for` loop:

```
for num in filter(is_odd, nums):
    print(num)
```

Again, you can accomplish the same thing with a list comprehension:

```
odd_nums = [num for num in nums if is_odd(num)]
```

❖ 1.5.3. Using Lambda Functions with map() and filter()

The `map()` and `filter()` functions are both often used with lambda functions, like this:

```
>>> nums1 = range(0, 10)
>>> nums2 = range(10, 0, -1)
>>> for multiple in map(lambda n: nums1[n] * nums2[n], range(10)):
...     print(multiple)
...
0
9
16
21
24
25
24
21
16
9
```

```
>>> for num in filter(lambda n: n % 2 == 1, range(10)):
...     print(num)
...
1
3
5
7
9
```

Let's just *keep* lambda

Some programmers, including Guido van Rossum, the creator of Python, dislike `lambda`, `filter()` and `map()`. These programmers feel that list comprehension can generally be used instead. However, other programmers love these functions and Guido eventually gave up the fight to remove them from Python. In February, 2006, he wrote:[1]

> *After so many attempts to come up with an alternative for lambda, perhaps we should admit defeat. I've not had the time to follow the most recent rounds, but I propose that we keep lambda, so as to stop wasting everybody's talent and time on an impossible quest.*

1. https://mail.python.org/pipermail/python-dev/2006-February/060415.html

You'll have to decide for yourself whether or not to use them.

1.6. Mutable and Immutable Built-in Objects

The difference between mutable objects, such as lists and dictionaries, and immutable objects, such as strings, integers, and tuples, may seem pretty straightforward: mutable objects can be changed; immutable objects cannot. But it helps to have a deeper understanding of how this can affect your code.

Consider the following code:

```
>>> v1 = 'A'
>>> v2 = 'A'
>>> v1 is v2
True
>>> list1 = ['A']
>>> list2 = ['A']
>>> list1 is list2
False
```


Immutable objects cannot be modified in place. Every time you "change" a string, you are actually creating a new string:

```
>>> name = 'Nat'
>>> id(name)
2613698710320
>>> name += 'haniel'
>>> id(name)
2613698710512
```

Notice the ids are different. It is impossible to modify an immutable object in place.

Lists, on the other hand, are mutable and can be modified in place. For example:

```
>>> v1 = [1, 2]
>>> v2 = v1
>>> id(v1) == id(v2)
True # Both variables point to the same list
>>> v1, v2
([1, 2], [1, 2])
>>> v2 += [3]
>>> v1, v2
([1, 2, 3], [1, 2, 3])
>>> id(v1) == id(v2)
True # Both variables still point to the same list
```

Notice that with lists, `v2` changes when we change `v1`. Both are pointing at the same list object, which is mutable. So, when we modify the `v2` list, we see the change in `v1`, because it points to the same object.

Be careful though. If you use the assignment operator, you will overwrite the old list and create a new list object:

```
>>> v1 = [1, 2]
>>> v2 = v1
>>> v1, v2
([1, 2], [1, 2])
>>> v1 = v1 + [3]
>>> v1, v2
([1, 2, 3], [1, 2])
```

Assigning `v1` explicitly with the assignment operator, rather than appending a value via `v1.append(3)`, results in a new object.

1.7. Sorting

❖ 1.7.1. Sorting Lists in Place

Python lists have a `sort()` method that sorts the list in place:

```
colors = ['red', 'blue', 'green', 'orange']
colors.sort()
```

The `colors` list will now contain:

```
['blue', 'green', 'orange', 'red']
```

The `sort()` method can take two keyword arguments: `key` and `reverse`.

reverse

The `reverse` argument is a boolean:


```
colors = ['red', 'blue', 'green', 'orange']
colors.sort(reverse=True)
```

The `colors` list will now contain:

```
['red', 'orange', 'green', 'blue']
```

key

The `key` argument takes a function to be called on each list item and performs the sort based on the result. For example, the following code will sort by word length:

```
colors = ['red', 'blue', 'green', 'orange']
colors.sort(key=len)
```

The `colors` list will now contain:

```
['red', 'blue', 'green', 'orange']
```

And the following code will sort by last name:

```
def get_lastname(name):
    return name.split()[-1]

people = ['George Washington', 'John Adams',
          'Thomas Jefferson', 'John Quincy Adams']
people.sort(key=get_lastname)
```

The `people` list will now contain:

```
[
    'John Adams',
    'John Quincy Adams',
    'Thomas Jefferson',
    'George Washington'
]
```


Note that John Quincy Adams shows up after John Adams in the result only because he shows up after him in the initial list. Our code as it stands does not take into account middle or first names.

Using Lambda Functions with key

If you don't want to create a new named function just to perform the sort, you can use a lambda function. For example, the following code would do the same thing as the code above without the need for the `get_lastname()` function:

```
people = ['George Washington', 'John Adams',,
          'Thomas Jefferson', 'John Quincy Adams']
people.sort(key=lambda name: name.split()[-1])
```

Combining key and reverse

The `key` and `reverse` arguments can be combined. For example, the following code will sort by word length in descending order:

```
colors = ['red', 'blue', 'green', 'orange']
colors.sort(key=len, reverse=True)
```

The `colors` list will now contain:

```
['orange', 'green', 'blue', 'red']
```


❖ 1.7.2. The sorted() Function

The built-in `sorted()` function requires an iterable as its first argument and can take `key` and `reverse` as keyword arguments. It works just like the list's `sort()` method except that:

1. It does **not** modify the iterable in place. Rather, it returns a new sorted list.
2. It can take any iterable, not just a list (but it always returns a list).

Exercise 4: Converting list.sort() to sorted(iterable)

15 to 25 minutes

In this exercise, you will convert all the examples of `sort()` we saw earlier to use `sorted()` instead.

1. Open `advanced-python-concepts/Exercises/sorting.py` in your editor.
2. The code in the first example has already been converted to use `sorted()`.
3. Convert all other code examples in the script.

Exercise Code 4.1: advanced-python-concepts/Exercises/sorting.py

```
1.   # Simple sort() method
2.   colors = ['red', 'blue', 'green', 'orange']
3.   # colors.sort()
4.   new_colors = sorted(colors) # This one has been done for you
5.   print(new_colors)
6.
7.   # The reverse argument:
8.   colors.sort(reverse=True)
9.   print(colors)
10.
11.  # The key argument:
12.  colors.sort(key=len)
13.  print(colors)
14.
15.  # The key argument with named function:
16.  def get_lastname(name):
17.      return name.split()[-1]
18.
19.  people = ['George Washington', 'John Adams',
20.            'Thomas Jefferson', 'John Quincy Adams']
21.  people.sort(key=get_lastname)
22.  print(people)
23.
24.  # The key argument with lambda:
25.  people.sort(key=lambda name: name.split()[-1])
26.  print(people)
27.
28.  # Combing key and reverse
29.  colors.sort(key=len, reverse=True)
30.  print(colors)
```


When you're done, run the file. The output should look like this:

```
['blue', 'green', 'orange', 'red']
['red', 'orange', 'green', 'blue']
['red', 'blue', 'green', 'orange']
['John Adams', 'John Quincy Adams', 'Thomas Jefferson', 'George Washington']
['John Adams', 'John Quincy Adams', 'Thomas Jefferson', 'George Washington']
['orange', 'green', 'blue', 'red']
```

Solution: advanced-python-concepts/Solutions/sorting.py

```
1.  # Simple sort() method
2.  colors = ['red', 'blue', 'green', 'orange']
3.  # colors.sort()
4.  new_colors = sorted(colors) # This one has been done for you
5.  print(new_colors)
6.
7.  # The reverse argument:
8.  # colors.sort(reverse=True)
9.  # print(colors)
10. new_colors = sorted(colors, reverse=True)
11. print(new_colors)
12.
13. # The key argument:
14. # colors.sort(key=len)
15. # print(colors)
16. new_colors = sorted(colors, key=len)
17. print(new_colors)
18.
19. # The key argument with named function:
20. def get_lastname(name):
21.     return name.split()[-1]
22.
23. people = ['George Washington', 'John Adams',
24.           'Thomas Jefferson', 'John Quincy Adams']
25. # people.sort(key=get_lastname)
26. # print(people)
27. new_people = sorted(people, key=get_lastname)
28. print(new_people)
29.
30. # The key argument with lambda function:
31. people = ['George Washington', 'John Adams',
32.           'Thomas Jefferson', 'John Quincy Adams']
33. # people.sort(key=lambda name: name.split()[-1])
34. # print(people)
35. new_people = sorted(people, key=lambda name: name.split()[-1])
36. print(new_people)
37.
38. # Combining key and reverse
39. # colors.sort(key=len, reverse=True)
40. # print(colors)
41. new_colors = sorted(colors, key=len, reverse=True)
42. print(new_colors)
```


1.8. Sorting Sequences of Sequences

When you sort a sequence of sequences, Python first sorts by the first element of each sequence, then by the second element, and so on. For example:

```
ww2_leaders = [
    ('Charles', 'de Gaulle'),
    ('Winston', 'Churchill'),
    ('Teddy', 'Roosevelt'), # Not a WW2 leader, but helps make point
    ('Franklin', 'Roosevelt'),
    ('Joseph', 'Stalin'),
    ('Adolph', 'Hitler'),
    ('Benito', 'Mussolini'),
    ('Hideki', 'Tojo')
]

ww2_leaders.sort()
```


The `ww2_leaders` list will be sorted by first name and then by last name. It will now contain:

```
[
    ('Adolph', 'Hitler'),
    ('Benito', 'Mussolini'),
    ('Charles', 'de Gaulle'),
    ('Franklin', 'Roosevelt'),
    ('Hideki', 'Tojo'),
    ('Joseph', 'Stalin'),
    ('Teddy', 'Roosevelt'),
    ('Winston', 'Churchill')
]
```

To change the order of the sort, use a lambda function:

```
ww2_leaders.sort(key=lambda leader: (leader[1], leader[0]))
```

The `ww2_leaders` list will now be sorted by last name and then by first name. It will now contain:

```
[
    ('Winston', 'Churchill'),
    ('Adolph', 'Hitler'),
    ('Benito', 'Mussolini'),
    ('Franklin', 'Roosevelt'),
    ('Teddy', 'Roosevelt'),
    ('Joseph', 'Stalin'),
    ('Hideki', 'Tojo'),
    ('Charles', 'de Gaulle')
]
```

It may seem strange that "de Gaulle" comes after "Tojo," but that is correct. Lowercase letters come after uppercase letters in sorting. To change the result, you can use the `lower()` function:

```
ww2_leaders.sort(key=lambda leader: (leader[1].lower(), leader[0]))
```

`ww2_leaders` will now contain:


```
[
    ('Winston', 'Churchill'),
    ('Charles', 'de Gaulle'),
    ('Adolph', 'Hitler'),
    ('Benito', 'Mussolini'),
    ('Franklin', 'Roosevelt'),
    ('Teddy', 'Roosevelt'),
    ('Joseph', 'Stalin'),
    ('Hideki', 'Tojo')
]
```

The preceding code can also be found in `advanced-python-concepts/Demos/sequence_of_sequences.py`.

❖ 1.8.1. Sorting Sequences of Dictionaries

You may often find data stored as lists of dictionaries:

```
from datetime import date
ww2_leaders = []
ww2_leaders.append(
    {'fname':'Winston', 'lname':'Churchill', 'dob':date(1889,4,20)}
)
ww2_leaders.append(
    {'fname':'Charles', 'lname':'de Gaulle', 'dob':date(1883,7,29)}
)
ww2_leaders.append(
    {'fname':'Adolph', 'lname':'Hitler', 'dob':date(1890,11,22)}
)
ww2_leaders.append(
    {'fname':'Benito', 'lname':'Mussolini', 'dob':date(1882,1,30)}
)
ww2_leaders.append(
    {'fname':'Franklin', 'lname':'Roosevelt', 'dob':date(1884,12,30)}
)
ww2_leaders.append(
    {'fname':'Joseph', 'lname':'Stalin', 'dob':date(1878,12,18)}
)
ww2_leaders.append(
    {'fname':'Hideki', 'lname':'Tojo', 'dob':date(1874,11,30)}
)
```


This data can be sorted using a lambda function similar to how we sorted lists of tuples:

```
ww2_leaders.sort(key=lambda leader: leader['dob'])
```

You can use this same technique to sort by a tuple:

```
ww2_leaders.sort(key=lambda leader: (leader['lname'], leader['fname']))
```

The preceding code can also be found in `advanced-python-concepts/Demos/sequence_of_dictionaries.py`.

`itemgetter()`

While the method shown above works fine, the `operator` module provides an `itemgetter()` method that performs this same task a bit faster. It works like this:

Demo 1.16: advanced-python-concepts/Demos/sorting_with_itemgetter.py

```
1.   from datetime import date
2.   from operator import itemgetter
3.
4.   def main():
     -------Lines 5 through 27 Omitted-------
28.      ww2_leaders.sort(key=itemgetter('dob'))
29.      print('First born:', ww2_leaders[0]['fname'])
30.
31.      ww2_leaders.sort(key=itemgetter('lname', 'fname'))
32.      print('First in Encyclopedia:', ww2_leaders[0]['fname'])
33.
34.  main()
```

1.9. Creating a Dictionary from Two Sequences

Follow these steps to make a dictionary from two lists using the first list for keys and the second list for values:

1. Use the built-in zip() function to make a list of two-element tuples from the two lists:

```
>>> courses = ['English', 'Math', 'Art', 'Music']
>>> grades = [96, 99, 88, 94]
>>> z = zip(courses, grades)
```

   This will result in a `zip` object.

2. Pass the `zip` object to the `dict()` constructor:

```
>>> course_grades = dict(z)
>>> course_grades
{'English': 96, 'Math': 99, 'Art': 88, 'Music': 94}
```

You can do the above in one step, like this:

```
course_grades = dict(zip(courses, grades))
```

This works with any type of sequence. For example, you could create a dictionary mapping letters to numbers like this:

```
>>> letter_mapping = dict(zip('abcdef', range(6)))
>>> letter_mapping
{'a': 0, 'b': 1, 'c': 2, 'd': 3, 'e': 4, 'f': 5}
```

1.10. Unpacking Sequences in Function Calls

Sometimes you'll have a sequence that contains the exact arguments a function needs. To illustrate, consider the following function:

```
import math
def distance_from_origin(a, b):
    return math.sqrt(a**2 + b**2)
```


The function expects two arguments, `a` and `b`, which are the x, y coordinates of a point. It uses the Pythagorean theorem to determine the distance the point is from the origin.

We can call the function like this:

```
c = distance_from_origin(3, 4)
```

But it would be nice to be able to call the function like this too:

```
point = (3, 4)
c = distance_from_origin(point)
```

However, that will cause an error because the function expects two arguments and we're only passing in one.

One solution would be to pass the individual elements of our point:

```
point = (3, 4)
c = distance_from_origin(point[0], point[1])
```

But Python provides an even easier solution. We can use an asterisk in the function call to unpack the sequence into separate elements:

```
point = (3, 4)
c = distance_from_origin(*point)
```


When you pass a sequence preceded by an asterisk into a function, the sequence gets *unpacked*, meaning that the function receives the individual elements rather than the sequence itself.

The preceding code can also be found in `advanced-python-concepts/Demos/unpacking_function_arguments.py`.

Exercise 5: Converting a String to a datetime.date Object

10 to 20 minutes

In this exercise, you will convert a string representing a date to a `datetime.date` object. This is the starting code:

Exercise Code 5.1: advanced-python-concepts/Exercises/converting_date_string_to_datetime.py

```
1.  import datetime
2.
3.  def str_to_date(str_date):
4.      # Write function
5.      pass
6.
7.  str_date = input('Input date as YYYY-MM-DD: ')
8.  date = str_to_date(str_date)
9.  print(date)
```


1. Open `advanced-python-concepts/Exercises/converting_date_string_to_datetime.py` in your editor.
2. The imported `datetime` module includes a `date()` method that can create a `date` object from three passed-in parameters: `year`, `month`, and `day`. For example:

   ```
   datetime.date(1776, 7, 4)
   ```

3. Write the code for the `str_to_date()` function so that it…
   A. Splits the passed-in string into a list of date parts. Each part should be an integer.
   B. Returns a `date` object created by passing the unpacked list of date parts to `datetime.date()`.

Solution:
advanced-python-concepts/Solutions/converting_date_string_to_datetime.py

```
1.  import datetime
2.
3.  def str_to_date(str_date):
4.      date_parts = [int(i) for i in str_date.split("-")]
5.      return datetime.date(*date_parts)
6.
7.  str_date = input('Input date as YYYY-MM-DD: ')
8.  date = str_to_date(str_date)
9.  print(date)
```

1.11. Modules and Packages

You have worked with different Python modules (e.g., `random` and `math`) and packages (e.g., `collections`). In general, it's not all that important to know whether a library you want to use is a module or a package, but there is a difference, and when you're creating your own, it's important to understand that difference.

❖ 1.11.1. Modules

A module is a single file. It can be made up of any number of functions and classes. You can import the whole module using:

```
import module_name
```

Or you can import specific functions or classes from the module using:

```
from module_name import class_or_function_name
```

For example, if you want to use the `random()` function from the `random` module, you can do so by importing the whole module or by importing just the `random()` function:

```
>>> import random
>>> random.random()
0.8843194837647139
>>> from random import random
>>> random()
0.251050616400771
```

As shown above, when you import the whole module, you must prefix the module's functions with the module name.

Every `.py` file is a module. When you build a module with the intention of making it available to other modules for importing, it is common to include a `_test()` function that runs tests when the module is run directly. For example, if you run `random.py`, which is in the `Lib` directory of your Python home, the output will look something like this:

```
2000 times random
0.003 sec, avg 0.500716, stddev 0.285239, min 0.000495333, max 0.99917

2000 times normalvariate
0.004 sec, avg 0.0061499, stddev 0.971102, min -2.86188, max 3.02266

2000 times lognormvariate
0.004 sec, avg 1.64752, stddev 2.12612, min 0.0310675, max 28.5174
…
```


Where is My Python Home?

To find your Python home, run the following code:

Demo 1.17: advanced-python-concepts/Demos/find_python_home.py

```
1.  import sys
2.  import os
3.
4.  python_home = os.path.dirname(sys.executable)
5.  print(python_home)
```

Open `random.py` in an editor and you will see it ends with this code:

```
if __name__ == '__main__':
  _test()
```

The `__name__` variable of any module that is imported holds that module's name. For example, if you import `random` and then print `random.__name__`, it will output "random". However, if you open `random.py`, add a line that reads `print(__name__)`, and run it, it will print "__main__". So, the `if` condition in the code above just checks to see if the file has been imported. If it hasn't (i.e., if it's running directly), then it will call the `_test()` function.

If you do not want to write tests, you could include code like this:

```
if __name__ == '__main__':
    print('This module is for importing, not for running directly.')
```

❖ 1.11.2. Packages

A package is a group of files (and possibly subfolders) stored in a directory that includes a file named *__init__.py*. The *__init__.py* file does not need to contain any code. Some libraries' *__init__.py* files have a simple comment like this:


```
# Dummy file to make this a package.
```

However, you can include code in the *__init__.py* file that will initialize the package. You can also (but do not have to) set a global `__all__` variable, which should contain a list of files to be imported when a file imports your package using `from package_name import *`. If you do not set the `__all__` variable, then that form of import will not be allowed, which may be just fine.

❖ 1.11.3. Search Path for Modules and Packages

The Python interpreter must locate the imported modules. When import is used within a script, the interpreter searches for the imported module in the following places sequentially:

1. The current directory (same directory as script doing the importing).
2. The library of standard modules.
3. The paths defined in `sys.path`.[2]

2. `sys.path` contains a list of strings specifying the search path for modules. The list is os-dependent. To see your list, import `sys`, and then output `sys.path`.

As you see, the steps involved in creating modules and packages for import are relatively straightforward. However, designing useful and easy-to-use modules and packages takes a lot of planning and thought.

Conclusion

In this lesson, you have learned several advanced techniques with sequences. You have also learned to do mapping and filtering, and to use lambda functions. Finally, you have learned how modules and packages are created.

LESSON 2
Regular Expressions

Topics Covered

☑ Understanding regular expressions.

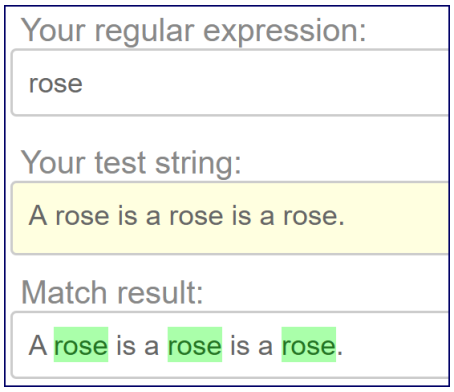
☑ Python's `re` module.

Introduction

Regular expressions are used to do pattern matching in many programming languages, including Java, PHP, JavaScript, C, C++, and Perl. We will provide a brief introduction to regular expressions and then we'll show you how to work with them in Python.

2.1. Regular Expression Tester

We will use the online regular expression testing tool at `https://pythex.org` to demonstrate and test our regular expressions. To see how it works, open the page in your browser:

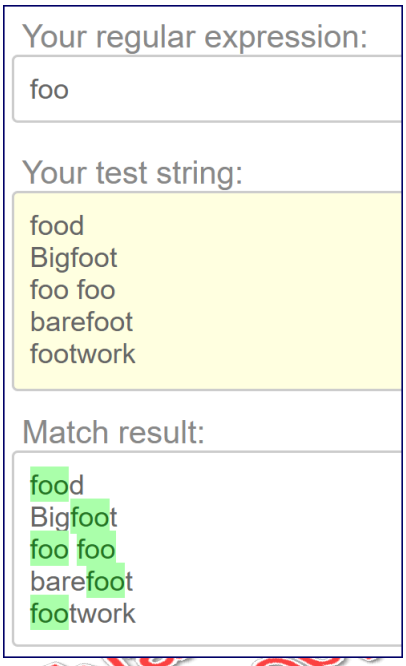


As shown in the screenshot:

1. Enter "rose" in the **Your regular expression** field.
2. Enter "A rose is a rose is a rose." in the **Your test string** field.
3. Notice the **Match result**. The parts of the string that match your pattern will be highlighted.

Usually, you will want to have the **MULTILINE** option selected so that each line will be tested individually.

In the **Your test string** field, you can test multiple strings:



These examples just find occurrences of a substring (e.g., "rose") in a string (e.g, "A rose is a rose is a rose."). But the power of regular expressions is in pattern matching. The best way to get a feel for them is to try them out. So, let's do that.

2.2. Regular Expression Syntax

Here we'll show the different symbols used in regular expressions. You should use `https://pythex.org` to test the patterns we show.

Start and End (^ $)

A caret (`^`) at the beginning of a regular expression indicates that the string being searched must start with this pattern.

- The pattern `^dog` can be found in "**dog**fish", but not in "bulldog" or "boondoggle".

A dollar sign (`$`) at the end of a regular expression indicates that the string being searched must end with this pattern.

- The pattern `dog$` can be found in "bull**dog**", but not in "dogfish" or "boondoggle".

Word Boundaries (\b \B)

Backslash-b (`\b`) denotes a word boundary. It matches a location at the beginning or end of a word. A word is a sequence of numbers, letters, and underscores. Any other character is considered a word boundary.

- The pattern `dog\b` matches the first but not the second occurence of "dog" in the phrase "The bull**dog** bit the dogfish."
- In the phrase "The dogfish bit the bull**dog**.", it only matches the second occurrence of "dog", because a period is a word boundary.

Backslash-B (`\B`) is the opposite of backslash-b (`\b`). It matches a location that is **not** a word boundary.

- The pattern `dog\B` matches the second but not the first occurence of "dog" in the phrase "The bulldog bit the **dog**fish."
- But in the phrase "The **dog**fish bit the bulldog.", it only matches the first occurrence of "dog".

Number of Occurrences (? + * {})

The following symbols affect the number of occurrences of the preceding character: `?`, `+`, `*`, and `{}`.

A question mark (`?`) indicates that the preceding character should appear zero or one times in the pattern.

- The pattern `go?ad` can be found in "g**o**ad" and "gad", but not in "g**oo**ad". Only zero or one "o" is allowed before the "a".

A plus sign (`+`) indicates that the preceding character should appear one or more times in the pattern.

- The pattern `go+ad` can be found in "g**o**ad", "g**oo**ad" and "g**ooo**ad", but not in "gad".

An asterisk (`*`) indicates that the preceding character should appear zero or more times in the pattern.

- The pattern `go*ad` can be found in "gad", "g**o**ad", "g**oo**ad" and "g**ooo**ad".

Curly braces with one parameter (`{n}`) indicate that the preceding character should appear exactly `n` times in the pattern.

- The pattern `fo{3}d` can be found in "f**ooo**d" , but not in "food" or "fooood".

Curly braces with two parameters (`{n1,n2}`) indicate that the preceding character should appear between `n1` and `n2` times in the pattern.

- The pattern `fo{2,4}d` can be found in "f**oo**d", "f**ooo**d" and "f**oooo**d", but not in "fod" or "foooood".

Curly braces with one parameter and an empty second parameter (`{n,}`) indicate that the preceding character should appear at least `n` times in the pattern.

- The pattern `fo{2,}d` can be found in "f**oo**d" and "f**oooooo**d", but not in "fod".

Common Characters (. \d \D \w \W \s \S)

A period (`.`) represents any character except a newline.

- The pattern `fo.d` can be found in "fo**o**d", "fo**a**d", "fo**9**d", and "fo d".

Backslash-d (`\d`) represents any digit. It is the equivalent of `[0-9]` (to be discussed soon).

- The pattern `fo\dd` can be found in "fo**1**d", "fo**4**d" and "fo**0**d", but not in "food" or "fodd".

Backslash-D (`\D`) represents any character except a digit. It is the equivalent of `[^0-9]` (to be discussed soon).

- The pattern `fo\Dd` can be found in "fo**o**d" and "fo**l**d", but not in "fo4d".

Backslash-w (`\w`) represents any word character (letters, digits, and the underscore (_)).

- The pattern `fo\wd` can be found in "fo**o**d", "fo_d" and "fo**4**d", but not in "fo*d".

Backslash-W (`\W`) represents any character except a word character.

- The pattern `fo\Wd` can be found in "fo*d", "fo**@**d" and "fo**.**d", but not in "food".

Backslash-s (`\s`) represents any whitespace character (e.g., space, tab, newline).

- The pattern `fo\sd` can be found in "fo d", but not in "food".

Backslash-S (`\S`) represents any character except a whitespace character.

- The pattern `fo\Sd` can be found in “fo*d”, “fo**o**d” and “fo**4**d”, but not in “fo d”.

Character Classes ([])

Square brackets (`[]`) are used to create a character class (or character set), which specifies a set of characters to match.

- The pattern `f[aeiou]d` can be found in “f**a**d” and “f**e**d”, but not in “food”, “fyd” or “fd”
    - `[aeiou]` matches an “a”, an “e”, an “i”, an “o”, or a “u”
- The pattern `f[aeiou]{2}d` can be found in “f**ae**d” and “f**eo**d”, but not in “fod”, “fold” or “fd”.
- The pattern `[A-Za-z]+` can be found twice in “**Webucator**, **Inc**.”, but not in “13066”.
    - `[A-Za-z]` matches any lowercase or uppercase letter.
    - `[A-Z]` matches any uppercase letter.
    - `[a-z]` matches any lowercase letter.
- The pattern `[1-9]+` can be found twice in “**13066**”, but not in “Webucator, Inc.”

Negation (^)

When used as the first character within a character class, the caret (`^`) is used for negation. It matches any characters **not** in the set.

- The pattern `f[^aeiou]d` can be found in “f**q**d” and “f**4**d”, but not in “fad” or “fed”.

Groups (())

Parentheses (()) are used to capture subpatterns and store them as groups, which can be retrieved later.

- The pattern `f(oo)?d` can be found in “f**oo**d” and “fd”, but not in “fod”.

The whole group can show up zero or one time.

Alternatives (|)

The pipe (`|`) is used to create optional patterns.

- The pattern `^web|or$` can be found in "**web**site", "educat**or**", and twice in "**web**ucat**or**", but not in "cobweb" or "orphan".

Escape Character (\ )

The backslash (`\`) is used to escape special characters.

- The pattern `fo\.d` can be found in "fo**.**d", but not in "food" or "fo4d".

❖ 2.2.1. Backreferences

Backreferences are special wildcards that refer back to a group within a pattern. They can be used to make sure that two subpatterns match. The first group in a pattern is referenced as `\1`, the second group is referenced as `\2`, and so on.

For example, the pattern `([bmpr])o\1` matches "**bob**cat", "ther**mom**eter", "**pop**ped", and "p**ror**ate".

A more practical example has to do with matching the delimiter in social security numbers. Examine the following regular expression:

```
^\d{3}([\- ]?)\d{2}([\- ]?)\d{4}$
```

Within the caret (^) and dollar sign ($), which are used to specify the beginning and end of the pattern, there are three sequences of digits, optionally separated by a hyphen or a space. This pattern will be matched in all of the following strings (and more):

- 123-45-6789
- 123 45 6789
- 123456789
- 123-45 6789
- 123 45-6789
- 123-456789

The last three strings are not ideal, but they do match the pattern. Backreferences can be used to make sure that the second delimiter matches the first delimiter. The regular expression would look like this:

```
^\d{3}([\- ]?)\d{2}\1\d{4}$
```

The `\1` refers back to the first subpattern. Only the first three strings listed above match this regular expression.

2.3. Python's Handling of Regular Expressions

In Python, you use the `re` module to access the regular expression engine. Here is a very simple illustration. Imagine you're looking for the pattern "r[aeiou]se" in the string "A rose is a rose is a rose."

1. Import the `re` module:

```
import re
```

2. Use the `compile()` method to make an object from the pattern that you can then use to search strings for pattern matches:

```
p = re.compile('r[aeiou]se')
```


3. Search the string for a match:

```
result = p.search('A rose is a rose is a rose.')
```

4. Print the result:

```
print(result)
```

This will print the following, showing that the result is a `match` object and that it found the match "rose" starting at index 2 and ending at index 6:

```
<_sre.SRE_Match object; span=(2, 6), match='rose'>
```

Compiling a regular expression pattern into an object is a good idea if you're going to reuse the expression throughout the program, but if you're just using it once or twice, you can use the module-level `search()` method, like this:

```
>>> result = re.search('r[aeiou]se', 'A rose is a rose is a rose.')
>>> result
<re.Match object; span=(2, 6), match='rose'>
```

Raw String Notation

Python uses the backslash character (`\`) to escape special characters. For example `\n` is a newline character. A call to `print('a\nb\nc')` will print the letters `a`, `b`, and `c` each on its own line:

```
>>> print('a\nb\nc')
a
b
c
```

If you actually want to print a backslash followed by an "n", you need to escape the backslash with another backslash, like this: `print('a\\nb\\nc')`. That will print the literal string "a\nb\nc":


```
>>> print('a\\nb\\nc')
a\nb\nc
```

Python provides another way of doing this. Instead of escaping all the backslashes, you can use *rawstring notation* by placing the letter "r" before the beginning of the string, like this: `print(r'a\nb\nc')`:

```
>>> print(r'a\nb\nc')
a\nb\nc
```

While this may not come in very handy in most areas of programming, it is very helpful when writing regular expression patterns. That is because the regular expression syntax also uses the backslash for special characters. If you don't use raw string notation, you may find your patterns filled with backslashes.

The takeaway: Always use raw string notation for your patterns.

Regular Expression Object Methods

1. `p.search(string)` – Finds the first substring that matches the pattern. Returns a `Match` object or `None`.

```
>>> p = re.compile(r'\W')
>>> p.search('andré@example.com')
<re.Match object; span=(5, 6), match='@'>
```

   This finds the first non-word character.

2. `p.match(string)` – Like `search()`, but the match must be found at the beginning of the string. Returns a `Match` object or `None`.

```
>>> p = re.compile(r'\W')
>>> p.match('andré@example.com') # Returns None
>>> p.match('@example.com')
<re.Match object; span=(0, 1), match='@'>
```


   This matches the first character if it is a non-word character. The first example returns `None` as the passed-in string begins with "a".

3. `p.fullmatch(string)` – Like `search()`, but the whole string must match. Returns a `Match` object or `None`.

```
>>> p = re.compile(r'[\w\.]+@example.com')
>>> p.match('andré@example.com')
<re.Match object; span=(0, 17), match='andré@example.com'>
```

   This matches a string made up of word characters and periods followed by "@example.com".

4. `p.findall(string)` – Finds all non-overlapping matches. Returns a list of strings.

```
>>> p = re.compile(r'\W')
>>> p.findall('andré@example.com')
['@', '.']
```

   This returns a list of all matches of non-word characters.

5. `p.split(string, maxsplit=0)` – Splits the string on pattern matches. If `maxsplit` is nonzero, limits splits to `maxsplit`. Returns a list of strings.

```
>>> p = re.compile(r'\W')
>>> p.split('andré@example.com')
['andré', 'example', 'com']
```

   This splits the string on non-word characters.

6. `p.sub(repl, string, count=0)` – Replaces all non-overlapping matches in `string` with `repl`. If `count` is nonzero, limits replacements to `count`. More details on `sub()` under **Using `sub()` with a Function** (see page 61). Returns a string.

All the methods that search a string for a pattern (`search()`, `match()`, `fullmatch()`, and `findall()`) include `start` and `end` parameters that indicate what positions in the string to start and end the search.

Groups

As discussed earlier, parentheses in regular expression patterns are used to capture groups. You can access these groups individually using a `match` object's `group()` method or all at once using its `groups()` method.

The `group()` method takes an integer[3] as an argument and returns a string:

- `match.group(0)` returns the whole match.
- `match.group(1)` returns the first group found.
- `match.group(2)` returns the second group found.
- And so on...

You can also get multiple groups at the same time returned as a tuple of strings by passing in more than one argument (e.g., `match.group(1, 2)`).

When nested parentheses are used in the pattern, the outer group is returned before the inner group. Here is an example that illustrates that:

3. Groups can also be named through a Python extension to regular expressions. For more information, see `https://docs.python.org/3/howto/regex.html#non-capturing-and-named-groups`.

```
>>> import re
>>> p = re.compile(r'(\w+)@(\w+\.(\w+))')
>>> match = p.match('andre@example.com')
>>> email = match.group(0)
>>> handle = match.group(1)
>>> domain = match.group(2)
>>> domain_type = match.group(3)
>>> print(email, handle, domain, domain_type, sep='\n')
andre@example.com
andre
example.com
com
```

Notice that "example.com" is group 2 and "com", which is nested within "example.com" is group 3.

And you can use the `groups()` method to get them all at once:

```
>>> print(match.groups())
('andre', 'example.com', 'com')
```


Flags

The `compile()` method takes an optional second argument: `flags`. The flags are constants that can be used individually or combined with a pipe (`|`).

```
re.compile(pattern, re.FLAG1|re.FLAG2)
```

The two most useful flags are (shortcut versions in parentheses):

1. `re.IGNORECASE` (`re.I`) – Makes the pattern case insensitive.
2. `re.MULTILINE` (`re.M`) – Makes `^` and `$` consider each line as an independent string.

Using sub() with a Function

The `sub()` method can either replace each match with a string or with the return value of a specified function. The function receives the match as an argument and must return a string that will replace the matched pattern. Here is an example:

Demo 2.1: regular-expressions/Demos/clean_cusses.py

```
1.   import re
2.   import random
3.
4.   def clean_cuss(match):
5.       # Get the whole match
6.       cuss = match.group(0)
7.       # Generate a random list of characters the length of cuss
8.       chars = [random.choice('!@#$%^&*') for letter in cuss]
9.       # Return the list joined into a string
10.      return ''.join(chars)
11.
12.  def main():
13.      pattern = r'\b[a-z]*(stupid|stinky|darn|shucks|crud|slob)[a-z]*\b'
14.      p = re.compile(pattern, re.IGNORECASE|re.MULTILINE)
15.      s = """Shucks! What a cruddy day I\'ve had.
16.  I spent the whole darn day with my slobbiest
17.  friend darning his STINKY socks."""
18.      result = p.sub(clean_cuss, s)
19.      print(result)
20.
21.  main()
```


Reading regular expressions is tricky. You have to think like a computer and parse it part by part:

Word boundary:

\b[a-z]*(stupid|stinky|darn|shucks|crud|slob)[a-z]*\b

0 or more lowercase letters:

\b**[a-z]***(stupid|stinky|darn|shucks|crud|slob)[a-z]*\b

Any one of the words delimited by the pipes (|):

\b[a-z]***(stupid|stinky|darn|shucks|crud|slob)**[a-z]*\b

0 or more lowercase letters:

\b[a-z]*(stupid|stinky|darn|shucks|crud|slob)**[a-z]***\b

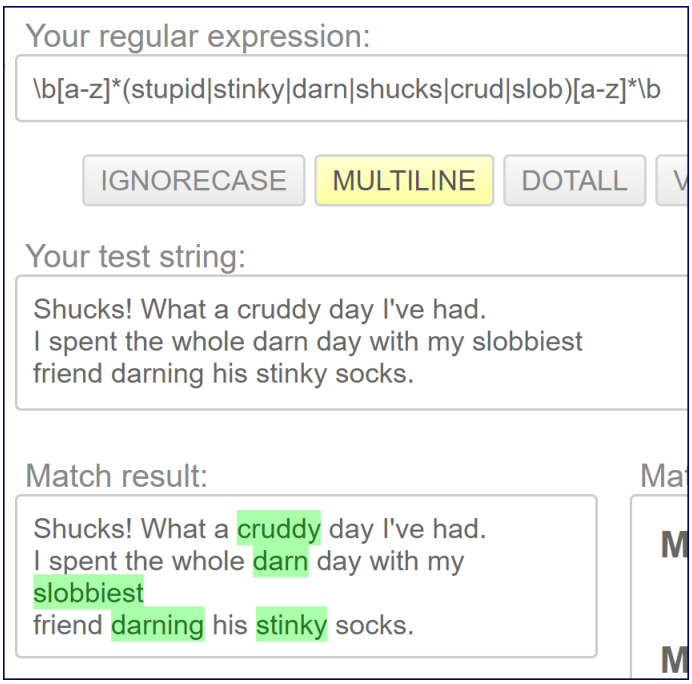
Word boundary:

\b[a-z]*(stupid|stinky|darn|shucks|crud|slob)[a-z]***\b**

Notice that we compile the pattern using the `re.IGNORECASE` and `re.MULTILINE` flags:

```
p = re.compile(pattern, re.IGNORECASE|re.MULTILINE)
```

The following screenshot shows the regular expression matches in pythex.org:



Run the file in the terminal to see that it replaces all those matches with a random string of characters:

```
…/regular-expressions/Demos> python clean_cusses.py
@$@$#!! What a !&!#%* day I've had.
I spent the whole &*$* day with my @$@!%$^*^
friend ^&#*&#& his ^!@*#@ socks.
```

In an earlier lesson (see page 24), we split the text of the *U.S. Declaration of Independence* on spaces to create a counter showing which words were used the most often. The resulting list looked like this:

```
[('PEOPLE', 13), ('STATES', 7), ('SHOULD', 5), ('INDEPENDENT', 5),
 ('AGAINST', 5), ('GOVERNMENT,', 4), ('ASSENT', 4),
 ('OTHERS', 4), ('POLITICAL', 3), ('POWERS', 3)]
```

In the following demo, we use a regular expression to split on any character that is not a capital letter:

Demo 2.2: regular-expressions/Demos/counter_re.py

```
1.   import re
2.   from collections import Counter
3.
4.   with open('Declaration_of_Independence.txt') as f:
5.       doi = f.read().upper()
6.
7.   word_list = [word for word in re.split('[^A-Z]', doi) if len(word) > 5]
8.
9.   c = Counter(word_list)
10.  print(c.most_common(10))
```


Because we use `upper()` to convert the whole text to uppercase, we can split on `[^A-Z]`. If we didn't know that there were only uppercase letters, we would have used `[^A-Za-z]` instead.

The new results are:

```
[('PEOPLE', 17), ('STATES', 12), ('GOVERNMENT', 7), ('POWERS', 6),
('BRITAIN', 6), ('SHOULD', 5), ('COLONIES', 5), ('INDEPENDENT', 5),
('AGAINST', 5), ('MANKIND', 4)]
```

Exercise 6: Green Glass Door

20 to 30 minutes

In this exercise, you will modify a function so that it uses a regular expression. But first, a little riddle:

The following items can pass through the green glass door:

1. puddles
2. mommies
3. aardvarks
4. balloons

The following items cannot pass through the green glass door:

1. ponds
2. moms
3. anteaters
4. kites

Knowing that, which of the following can pass through the green glass door?

1. bananas
2. apples
3. pears
4. grapes
5. cherries

Did you figure it out? The two that can pass are a**pp**les and che**rr**ies. Any word with a double letter can pass through the gr**ee**n gla**ss** d**oo**r.

Now, take a look at the following code:

Exercise Code 6.1: regular-expressions/Exercises/green_glass_door.py

```
1.  def green_glass_door(word):
2.      prev_letter = ''
3.      for letter in word:
4.          letter = letter.upper()
5.          if letter == prev_letter:
6.              return True
7.          prev_letter = letter
8.      return False
9.
10. fruits = ['banana', 'apple', 'pear', 'grape', 'cherry',
11.           'persimmons', 'orange', 'passion fruit']
12.
13. for fruit in fruits:
14.     if green_glass_door(fruit):
15.         print(f'YES! {fruit} can pass through the green glass door.')
16.     else:
17.         print(f'NO! {fruit} cannot pass through the green glass door.')
```


Study the code, paying particular attention to the `green_glass_door()` function. Your job is to rewrite that function to use a regular expression. Don't forget to import `re`.

Solution: regular-expressions/Solutions/green_glass_door.py

```
1.  import re
2.
3.  def green_glass_door(word):
4.      pattern = re.compile(r'(.)\1')
5.      return pattern.search(word)
6.
7.  fruits = ['banana', 'apple', 'pear', 'grape', 'cherry',
8.            'persimmons', 'orange', 'passion fruit']
9.
10. for fruit in fruits:
11.     if green_glass_door(fruit):
12.         print(f'YES! {fruit} can pass through the green glass door.')
13.     else:
14.         print(f'NO! {fruit} cannot pass through the green glass door.')
```

The first part of the pattern matches any character. It uses parentheses to create a group:

```
pattern = re.compile(r'(.)\1')
```


The second part of the pattern uses a backreference to match the first group: that is, the character matched by `(.)`:

```
pattern = re.compile(r'(.)\1')
```

The function then returns the result of searching the string for that pattern:

```
return pattern.search(word)
```

That will either return a `Match` object, which evaluates to `True`, or it will return `None`, which evaluates to `False`.

Conclusion

In this lesson, you have learned how to work with regular expressions in Python. To learn more about regular expressions, see Python's Regular Expression HOWTO (https://docs.python.org/3/howto/regex.html).

LESSON 3
Working with Data

Topics Covered

☑ Data stored in a relational database.

☑ Data stored in a CSV file.

☑ Data from a web page.

☑ HTML, XML, and JSON.

☑ Accessing an API.

Introduction

Data is stored in many different places and in many different ways. In this lesson, you'll learn about the Python modules that help you access data.

3.1. Virtual Environment

In this lesson, you will be installing some libraries. So as not to mess up your standard environment, you should create a virtual environment and work within it the entire lesson:

1. Open a terminal at `working-with-data` and run the following command:

   ```
   …/Python/working-with-data> python -m venv .venv
   ```

   This will create and populate a new `.venv` directory.

2. Activate the new virtual environment:

Windows

```
.venv/Scripts/activate
```

Mac / Linux

```
source .venv/bin/activate
```

Your prompt should now be prefaced with "(.venv)".

3. Throughout this lesson, you should work in the virtual directory. So, if you deactivate it to do other work (or play), be sure to reactivate it when you're ready to proceed.

3.2. Relational Databases

In this lesson, we will be working with MySQL and SQLite, but Python is able to connect to all the commonly used databases, including PostgreSQL, Microsoft SQL Server, and Oracle. All implementations for working with different relational databases should follow PEP 0249 -- Python Database API Specification v2.0[4], which we will describe shortly.

SQL

SQL stands for **S**tructured **Q**uery **L**anguage and is pronounced either *ess-que-el* or *sequel*. It is the language used by relational database management systems (RDBMS) to access and manipulate data and to create, structure, and destroy databases and database objects. In this lesson, we will mostly be concerned with `SELECT` statements. `SELECT` statements are used to pull data out of one or more database tables. The basic syntax is as follows:

```
SELECT column_name, column_name, column_name, …
FROM table_name
WHERE conditions
ORDER BY column_name
LIMIT NUM;
```

If you are new to SQL, focus on the column names after the `SELECT` keyword. Those are the variable names that Python will use to access the data.

4. https://www.python.org/dev/peps/pep-0249/

We will also use `CREATE TABLE` and `INSERT` statements to create tables and insert data into them. Don't worry too much about that SQL. The important thing to know is that Python is sending the statements to the database to be executed.

❖ 3.2.1. Lahman's Baseball Database

The database we will use for our examples is Lahman's Baseball Database[5], which includes a huge amount of data on Major League Baseball from 1871 to the present.

We have a hosted version of the MySQL database on Amazon. The connection information is as follows:

- **host**: lahman.csw1rmup8ri6.us-east-1.rds.amazonaws.com
- **user**: python
- **password**: python
- **database**: lahmansbaseballdb

Local Version of the MySQL Database

If you have MySQL installed locally and wish to create a local version of the MySQL database, see `https://github.com/WebucatorTraining/lahman-baseball-mysql` for instructions.

We will be using Oracle's `mysql-connector-python`[6] to connect Python to MySQL. Install `mysql-connector-python` in your virtual environment by running:

```
pip install mysql-connector-python
```

❖ 3.2.2. PEP[7] 0249 -- Python Database API Specification v2.0

PEP 0249 defines an API for Python interfaces that work with databases. Generally, you follow the following steps to pull data from a database (be sure to open a Python terminal and follow along):

5. http://www.seanlahman.com/baseball-archive/statistics/
6. https://pypi.org/project/mysql-connector-python/
7. PEP stands for Python Enhancement Proposal.

1. Import a Python Database API-2.0-compliant interface.

```
import mysql.connector
```

2. Open a connection to the database.

```
connection = mysql.connector.connect(
    host='lahman.csw1rmup8ri6.us-east-1.rds.amazonaws.com',
    user='python',
    passwd='python',
    db='lahmansbaseballdb'
)
```

3. Write your query. For example, this query will get the first and last names, weight, and year of debut for the heaviest five people in the `people` table:

```
query = """SELECT nameFirst, nameLast, weight, year(debut)
  FROM people
  ORDER BY weight DESC
  LIMIT 5
"""
```


4. Create a cursor for the connection:

```
cursor = connection.cursor()
```

5. Use the cursor to execute one or more queries:

```
cursor.execute(query)
```

6. Get the results of the query/queries from the cursor:

```
results = cursor.fetchall()
```

7. Do something with the results. Here we just output the results:

```
results
```

8. Close the cursor:

```
cursor.close()
```

9. Close the connection to the database:

```
connection.close()
```

Here is the complete code:

Demo 3.1: working-with-data/Demos/fetch_all.py

```
1.  # Be sure to install via pip install mysql-connector-python
2.  import mysql.connector
3.
4.  connection = mysql.connector.connect(
5.      host='lahman.csw1rmup8ri6.us-east-1.rds.amazonaws.com',
6.      user='python',
7.      passwd='python',
8.      db='lahmansbaseballdb'
9.  )
10.
11. query = """SELECT nameFirst, nameLast, weight, year(debut)
12.            FROM people
13.            ORDER BY weight DESC
14.            LIMIT 5"""
15.
16. cursor = connection.cursor()
17. cursor.execute(query)
18. results = cursor.fetchall()
19.
20. cursor.close()
21. connection.close()
22.
23. print(results)
```


This will output a list of tuples, one tuple for each record returned:

```
[
('Walter', 'Young', 320, 2005),
('Jumbo', 'Diaz', 315, 2014),
('CC', 'Sabathia', 300, 2001),
('Dmitri', 'Young', 295, 1996),
('Jumbo', 'Brown', 295, 1925)
]
```

Cursor Methods

These are the most common `cursor` methods:

1. `cursor.execute(operation [, parameters])` – Prepares and executes a database query or command. The returned value varies by implementation.
2. `cursor.executemany(operation, seq_of_parameters)` – Prepares a database query or command and executes it once for each item in `seq_of_parameters`. This is usually used for `INSERT` and `UPDATE` statements, not for queries that return result sets. The returned value varies by implementation.
3. `cursor.fetchone()` – Fetches the next row of a result set. Returns a single row of data.
4. `cursor.fetchmany(n=cursor.arraysize)` – Fetches the next `n` rows of a result set. `cursor.arraysize` defaults to `1`. Returns a list of data rows.
5. `cursor.fetchall()` – Fetches all data rows of a result set. Returns a list of data rows.

When to Use fetchone()

As a general rule, after executing a `SELECT` statement, you will use `cursor.fetchall()` to fetch all the results. If you want a limited number of records, you should use SQL to set that limit.

One exception to this rule is when you know you are just getting one record from the database. In the following example, we limit the number of records to one by selecting on the primary key field:

Demo 3.2: working-with-data/Demos/fetch_one.py

```
1.  import mysql.connector
2.
3.  connection = mysql.connector.connect(
4.      host='lahman.csw1rmup8ri6.us-east-1.rds.amazonaws.com',
5.      user='python',
6.      passwd='python',
7.      db='lahmansbaseballdb'
8.  )
9.
10. query = """SELECT nameFirst, nameLast, birthCity, birthState, birthYear
11.            FROM people
12.            WHERE playerID = 'jeterde01';"""
13.
14. cursor = connection.cursor()
15. cursor.execute(query)
16. result = cursor.fetchone()
17.
18. if result:
19.     player_name = result[0] + ' ' + result[1]
20.     birth_place = result[2] + ', ' + result[3]
21.     birth_year = result[4]
22.     print(f'{player_name} was born in {birth_place} in {birth_year}.')
23. else:
24.     print('No records returned.')
25.
26. cursor.close()
27. connection.close()
```


The result is a single tuple, which we use to create more meaningfully named variables, and then output:

```
Derek Jeter was born in Pequannock, NJ in 1974.
```

Notice that we use an `if` condition to check to make sure a result was returned just in case there are no players in the `people` table with that `playerID`.

❖ 3.2.3. Returning Dictionaries instead of Tuples

According to PEP 0249, the cursor fetch methods must return a single sequence of values (for one row) or a sequence of sequences (for multiple rows). By default, `mysql-connector-python` returns a tuple (for `fetchone()`) and a list of tuples (for `fetchmany()` and `fetchall()`), but you can change the cursor type to a dictionary by passing in `dictionary=True` to the `cursor()` method:

```
cursor = connection.cursor(dictionary=True)
```

For the query getting the five heaviest baseball players of all time, this would change the result from a list of tuples to a list of dictionaries. Give it a try:

1. Open `working-with-data/Demos/fetch_all.py` in your editor.
2. Pass `dictionary=True` to the `connection.cursor()` method:

```
cursor = connection.cursor(dictionary=True)
```

3. Run the file. It will output something like this (though not laid out so nicely[8]):

```
[
    {
        'nameFirst': 'Walter',
        'nameLast': 'Young',
        'weight': 320,
        'year(debut)': 2005
    },
    {
        'nameFirst': 'Jumbo',
        'nameLast': 'Diaz',
        'weight': 315,
        'year(debut)': 2014
    },
    {
        'nameFirst': 'CC',
        'nameLast': 'Sabathia',
        'weight': 300,
        'year(debut)': 2001
    },
    {
        'nameFirst': 'Dmitri',
        'nameLast': 'Young',
        'weight': 295,
        'year(debut)': 1996
    },
    {
        'nameFirst': 'Jumbo',
        'nameLast': 'Brown',
        'weight': 295,
        'year(debut)': 1925
    }
]
```


A big advantage of having the data returned as dictionaries is that you can then reference the columns by name instead of by position. The code in the following file illustrates this:

8. Check out the `pprint` library at `https://docs.python.org/3/library/pprint.html` to see how you can pretty print data results. We have an example at `working-with-data/Demos/pretty_print.py`.

Demo 3.3: working-with-data/Demos/fetch_one_as_dict.py

```
     -------Lines 1 through 9 Omitted-------
10.  query = """SELECT nameFirst, nameLast, birthCity, birthState, birthYear
11.             FROM people
12.             WHERE playerID = 'jeterde01';"""
13.
14.  cursor = connection.cursor(dictionary=True)
15.  cursor.execute(query)
16.  result = cursor.fetchone()
17.
18.  if result:
19.      player_name = result['nameFirst'] + ' ' + result['nameLast']
20.      birth_place = result['birthCity'] + ', ' + result['birthState']
21.      birth_year = result['birthYear']
22.      print(f'{player_name} was born in {birth_place} in {birth_year}.')
     -------Lines 23 through 27 Omitted-------
```

Notice that we can reference the columns as `result['nameFirst']`, `result['nameLast']`, etc. That's much more readable than `result[0]`, `result[1]`, etc.

3.3. Passing Parameters

Often, your Python code won't know some of the values in the SQL query until execution time. For example, we could write a program that allowed users to find out which five players had the most home runs in a given year. The SQL query for that would look like this:

```
SELECT p.nameFirst, p.nameLast, b.HR, t.name AS team, b.yearID
FROM batting b
    JOIN people p ON p.playerID = b.playerID
    JOIN teams t ON t.ID = b.team_ID
WHERE b.yearID = 1950
ORDER BY b.HR DESC
LIMIT 5;
```

But we need to make the year a variable.

The Wrong Way

You might be tempted to do this with a Python variable like this:

Don't do this!

```
year_id = int(input('Enter a year: '))

query = """SELECT p.nameFirst, p.nameLast, b.HR, t.name AS team, b.yearID
    FROM batting b
        JOIN people p ON p.playerID = b.playerID
        JOIN teams t ON t.ID = b.team_ID
    WHERE b.yearID = {year_id}
    ORDER BY b.HR DESC
    LIMIT 5;"""
```

In this case, Python creates the whole query as a string and sends it to the database to run.

There are multiple problems with the above approach, but the biggest one is that it opens the database to hacking. A savvy and nefarious person could try to pass in a value to `year_id` that ended one query and started another that either sought to extract extra data (e.g., passwords) from the database or tried to wreak havoc on your database by updating or deleting records.

The Right Way

The right way to construct a SQL query is to pass parameters to the database and let it do the work of constructing the query. This is beneficial for at least a couple of reasons:

1. It mitigates the security risk. The database can check the passed-in parameters to make sure that they match the data types of the corresponding columns. Any code that tried to end one query and start another would get rejected.
2. It allows the database to compile the query so that it can reuse it with different passed-in parameters without recompiling every time.

Different databases use different placeholders for parameters:

1. MySQL and PostgreSQL use `%s`.
2. Oracle[9] uses a `:` followed by a variable name or index.
3. SQL Server and SQLite use a `?`.

9. https://cx-oracle.readthedocs.io/en/latest/user_guide/bind.html

The following demo uses MySQL:

Demo 3.4: working-with-data/Demos/homerun_leaders_mysql.py

```
1.  import mysql.connector
2.
3.  def main():
4.      connection = mysql.connector.connect(
5.          host='lahman.csw1rmup8ri6.us-east-1.rds.amazonaws.com',
6.          user='python',
7.          passwd='python',
8.          db='lahmansbaseballdb'
9.      )
10.
11.     cursor = connection.cursor(prepared=True)
12.
13.     query = """SELECT p.nameFirst, p.nameLast, b.HR,
14.                     t.name AS team, b.yearID
15.                 FROM batting b
16.                     JOIN people p ON p.playerID = b.playerID
17.                     JOIN teams t ON t.ID = b.team_ID
18.                 WHERE b.yearID = %s
19.                 ORDER BY b.HR DESC
20.                 LIMIT 5;"""
21.
22.     checking = True
23.     while checking:
24.         year_id = int(input('Enter a year (0 to quit): '))
25.         if year_id == 0:
26.             break
27.
28.         cursor.execute(query, [year_id])
29.         results = cursor.fetchall()
30.
31.         for i, result in enumerate(results , 1):
32.             row = dict(zip(cursor.column_names, result))
33.             name = f"{row['nameFirst']} {row['nameLast']}"
34.             print(f"{i}. {name}: {row['HR']}")
35.
36.     cursor.close()
37.     connection.close()
38.
39. main()
```


Notice the `%s` parameter on line 18 and how a value for it (`year_id`) is passed in on line 28:

```
cursor.execute(query, [year_id])
```

Run the program. You will see that it prompts the user repeatedly for a year and then retrieves and prints out the five leading home-run hitters for that year.

Things to notice:

1. We pass `prepared=True` to `connection.cursor()`. This tells MySQL to create a `MySQLCursorPrepared` cursor, which is the type of cursor it uses to prepare SQL queries.
2. Unfortunately, you cannot use `dictionary=True` and `prepared=True` together, so we zip the `cursor.column_names` with each returned `result` to create a dictionary:

```
for i, result in enumerate(results , 1):
    row = dict(zip(cursor.column_names, result))
    name = f"{row['nameFirst']} {row['nameLast']}"
    print(f"{i}. {name}: {row['HR']}")
```

This allows us to access the values by name.

The `zip()` function was covered in **Creating a Dictionary from Two Sequences** in the **Advanced Python Concepts** lesson (see page 42).

3.4. SQLite[10]

SQLite is a server-less SQL database engine. Each SQLite database is stored in a single file that can easily be transported between computers. While SQLite is not as robust as enterprise relational database management systems, it works great for local databases or databases that don't have large loads.

Download the SQLite version of the Database

Webucator maintains a SQLite version of Lahman's Baseball Database at `https://github.com/WebucatorTraining/lahman-baseball-mysql/blob/master/lahmansbaseballdb.sqlite?raw=true`.

Download `lahmansbaseballdb.sqlite` into the `working-with-data/data` folder.

10. See `https://docs.python.org/3/library/sqlite3.html` for documentation on Python's sqlite3 library and `https://www.sqlite.org` for documentation on SQLite itself.

Python's `sqlite3` module conforms to PEP 0249.

The following code shows how to create the same home-run-leader program, but this time we connect to a SQLite database instead of MySQL:

Demo 3.5: working-with-data/Demos/homerun_leaders_sqlite.py

```
1.  import sqlite3
2.  from pathlib import Path
3.
4.  def main():
5.      db = Path("../data/lahmansbaseballdb.sqlite")
6.      if not db.exists():
7.          print(
8.            "You have to download the database from https://github.com/Webucator ↵↵
              Training/lahman-baseball-mysql/blob/master/lahmansbase ↵↵
              balldb.sqlite?raw=true and save it in the data folder."
9.          )
10.         return
11.     connection = sqlite3.connect(db)
12.     connection.row_factory = sqlite3.Row
13.
14.     cursor = connection.cursor()
15.
16.     query = """SELECT p.nameFirst, p.nameLast, b.HR,
17.                     t.name AS team, b.yearID
18.                 FROM batting b
19.                     JOIN people p ON p.playerID = b.playerID
20.                     JOIN teams t ON t.ID = b.team_ID
21.                 WHERE b.yearID = ?
22.                 ORDER BY b.HR DESC
23.                 LIMIT 5;"""
24.
25.     checking = True
26.     while checking:
27.         year_id = int(input('Enter a year (0 to quit): '))
28.         if year_id == 0:
29.             break
30.
31.         cursor.execute(query, [year_id])
32.         results = cursor.fetchall()
33.
34.         for i, result in enumerate(results , 1):
35.             name = f"{result['nameFirst']} {result['nameLast']}"
36.             print(f"{i}. {name}: {result['HR']}")
37.
38.     cursor.close()
39.     connection.close()
40.
41. main()
```

Things to notice:

1. We connect to a SQLite database stored in `working-with-data/data`.
2. By default, data is returned as a tuple, so you have to access values by index. Setting `connection.row_factory` to `sqlite3.Row` allows you to access values in returned records by column name or index. It is analagous to passing `dictionary=True` to `connection.cursor()` when using `mysql-connector-python`.
3. SQLite uses question marks as placeholders for parameters.
4. Everything else is done the same as it was done with MySQL.

Exercise 7: Querying a SQLite Database

10 to 15 minutes

In this exercise, you will use your knowledge of PEP 0249 to connect to and query the `lahmansbaseballdb.sqlite` database.

1. Open `working-with-data/Exercises/querying_a_sqlite_database.py` in your editor.
2. The connection to the SQLite database has already been made and the query has been written. We have also included the following line, which allows you to access the values by column name:

```
connection.row_factory = sqlite3.Row
```

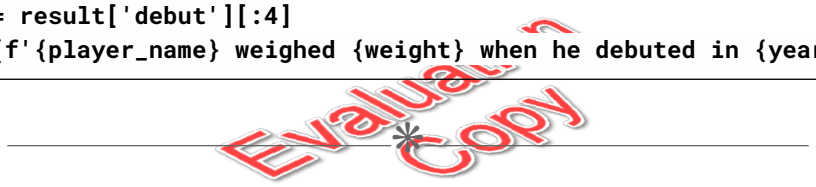

3. Add code that runs the query and assigns the results to the `results` variable.
4. Print out the results using a for loop, so that it outputs:

```
Walter Young weighed 320 when he debuted in 2005.
Jumbo Diaz weighed 315 when he debuted in 2014.
CC Sabathia weighed 300 when he debuted in 2001.
Jumbo Brown weighed 295 when he debuted in 1925.
Dmitri Young weighed 295 when he debuted in 1996.
```

   Note that the `debut` field is returned as a string in the format `YYYY-MM-DD`. You could use the `datetime` module to get at the year, or you can just use slicing.
5. Don't forget to close your cursor and connection.

Solution: working-with-data/Solutions/querying_a_sqlite_database.py

```
1.  import sqlite3
2.  connection = sqlite3.connect('../data/lahmansbaseballdb.sqlite')
3.  connection.row_factory = sqlite3.Row
4.
5.  query = """SELECT nameFirst, nameLast, weight,
6.                    debut AS debut
7.             FROM people
8.             ORDER BY weight DESC
9.             LIMIT 5"""
10.
11. cursor = connection.cursor()
12. cursor.execute(query)
13. results = cursor.fetchall()
14. cursor.close()
15. connection.close()
16.
17. for result in results:
18.     player_name = result['nameFirst'] + ' ' + result['nameLast']
19.     weight = result['weight']
20.     year = result['debut'][:4]
21.     print(f'{player_name} weighed {weight} when he debuted in {year}.')
```



3.5. SQLite Database in Memory

Python allows you to create in-memory databases with SQLite. This can be useful when you have a lot of data in a tab-delimited file that you want to query using SQL, but don't want to maintain as a database file as well. To create a connection to an in-memory database, use the following code:

```
connection = sqlite3.connect(':memory:')
```

In-memory SQLite Database

For the rest of the database section, we will work with an in-memory SQLite database, but the concepts apply to all databases.

You then create your tables with `CREATE TABLE` statements and populate them with `INSERT` statements. For example:

Demo 3.6: working-with-data/Demos/sqlite3_in_memory_tables.py

```
1.  import sqlite3
2.  connection = sqlite3.connect(':memory:')
3.  cursor = connection.cursor()
4.
5.  create = """CREATE TABLE beatles (
6.                  'fname' text,
7.                  'lname' text,
8.                  'nickname' text
9.              )"""
10.
11. cursor.execute(create)
12.
13. members = [
14.     ('John', 'Lennon', 'The Smart One'),
15.     ('Paul', 'McCartney', 'The Cute One'),
16.     ('George', 'Harrison', 'The Funny One'),
17.     ('Ringo', 'Starr', 'The Quiet One')
18. ]
19.
20. insert = 'INSERT INTO beatles VALUES (?, ?, ?)'
21.
22. # Loop through the members list, inserting each member
23. for member in members:
24.     cursor.execute(insert, member)
25.
26. select = 'SELECT fname, lname, nickname FROM beatles'
27. cursor.execute(select)
28.
29. results = cursor.fetchall()
30. cursor.close()
31. connection.close()
32.
33. print(results)
```


❖ 3.5.1. Executing Multiple Queries at Once

You might have noticed that, to insert the records, we looped through a list of lists, inserting one record at a time using:

```
cursor.execute(insert, member)
```

A better way of doing this is to use the `executemany()` method, which takes two arguments:

1. The query to run, which usually includes some placeholders to replace with passed-in values.
2. A sequence of sequences, each of which contains the values with which to replace the placeholders.

The query will run once for each sequence in the sequence of sequences. Different databases may implement this in different ways; however, the Python code should work with any database as long as your using a Python Database API-2.0-compliant interface.

Here is an example:

Demo 3.7: working-with-data/Demos/sqlite3_multiple_queries_at_once.py


```
     -------Lines 1 through 12 Omitted-------
13.  members = [
14.      ('John', 'Lennon', 'The Smart One'),
15.      ('Paul', 'McCartney', 'The Cute One'),
16.      ('George', 'Harrison', 'The Funny One'),
17.      ('Ringo', 'Starr', 'The Quiet One')
18.  ]
19.
20.  insert = 'INSERT INTO beatles VALUES (?, ?, ?)'
21.  cursor.executemany(insert, members)
     -------Lines 22 through 30 Omitted-------
```

Exercise 8: Inserting File Data into a Database

20 to 30 minutes

In this exercise, you will use the data from a text file to populate a database table.

Open `working-with-data/data/states.txt` in your editor. The file has 52 lines of data (50 states and Washington, D.C. and Puerto Rico). Each line contains three pieces of data[11] separated by tabs:

1. State Name
2. Population in 2020
3. Population in 2000

For example, the line for California reads:

```
California    39,631,049    37,254,523
```

Here is the starting code:

11. The state populations are based on data from `https://en.wikipedia.org/wiki/List_of_U.S._states_and_territories_by_population`. 2020 numbers are extrapolated.

Exercise Code 8.1: working-with-data/Exercises/inserting_file_data_into_a_database.py

```
1.   import sqlite3
2.   connection = sqlite3.connect(':memory:')
3.   connection.row_factory = sqlite3.Row
4.
5.   # Create the cursor.
6.
7.   create = """CREATE TABLE states (
8.                   'state' text,
9.                   'pop2020' integer,
10.                  'pop2000' integer
11.              )"""
12.
13.  # Execute the create statement.
14.
15.  insert = 'INSERT INTO states VALUES (?, ?, ?)'
16.
17.  # Create a list of tuples from the data in '../data/states.txt'.
18.
19.  # Insert the data into the database.
20.
21.  select = """SELECT state,
22.              CAST((pop2020*1.0/pop2000) * pop2020 AS INTEGER) AS pop2040
23.              FROM states ORDER BY pop2040 DESC"""
24.
25.  # Execute the select statement.
26.
27.  # Fetch the rows into a variable.
28.
29.  # Close the cursor and connection.
30.
31.  results = cursor.fetchall()
32.  cursor.close()
33.  connection.close()
34.
35.  # Print out the results.
```


Open `working-with-data/Exercises/inserting_file_data_into_a_database.py` in your editor. A connection to an in-memory database has already been established and the SQL statements for creating the table, inserting the records, and selecting the data have been written and stored in variables.

Your job is to:

1. Create a cursor.
2. Run the `CREATE` statement.
3. Get the data from the `states.txt` file into a list of 52 three-element tuples. **Note** that you will have to remove the commas from the population numbers so that they are valid integers.
4. Insert the rows using the list of tuples (i.e., the sequence of sequences) you just created.
5. Run the `SELECT` statement. This returns two columns: the state and the projected population in 2040. The column names are `state` and `pop2040`.
6. Fetch the results and output a sentence for each row (e.g., "The projected 2040 population of California is 42,159,177.").
7. Don't forget to close your cursor and connection.

Solution: working-with-data/Solutions/inserting_file_data_into_a_database.py

```
1.  import sqlite3
2.  connection = sqlite3.connect(':memory:')
3.  connection.row_factory = sqlite3.Row
4.
5.  cursor = connection.cursor()
6.
7.  create = """CREATE TABLE states (
8.                  'state' text,
9.                  'pop2020' integer,
10.                 'pop2000' integer
11.             )"""
12.
13. cursor.execute(create)
14.
15. insert = 'INSERT INTO states VALUES (?, ?, ?)'
16.
17. data = []
18. with open('../data/states.txt') as f:
19.     for line in f.readlines():
20.         state_data = line.split('\t')
21.         tpl_state_data = (state_data[0],
22.                           int(state_data[1].replace(',','')),
23.                           int(state_data[2].replace(',','')))
24.
25.         data.append(tpl_state_data)
26.
27. cursor.executemany(insert, data)
28.
29. select = """SELECT state,
30.             CAST((pop2020*1.0/pop2000) * pop2020 AS INTEGER) AS pop2040
31.             FROM states ORDER BY pop2040 DESC"""
32.
33. cursor.execute(select)
34.
35. results = cursor.fetchall()
36. cursor.close()
37. connection.close()
38.
39. for record in results:
40.     state = record['state']
41.     pop2040 = record['pop2040']
42.     print(f'The projected 2040 population of {state} is {pop2040:,}.')
```

3.6. Drivers for Other Databases

We have used `mysql-connector-python` to connect Python to MySQL and Python's `sqlite3` module to connect to SQLite. We recommend the following drivers for other major databases:

- PostgreSQL: `psycopg2` (`https://pypi.org/project/psycopg2/`)
- SQL Server: `pyodbc` (`https://pypi.org/project/pyodbc/`)
- Oracle: `cx-Oracle` (`https://pypi.org/project/cx-Oracle/`)

All of these comply with PEP 0249.

3.7. CSV

CSV (for "Comma Separated Values") is a format commonly used for sharing data between applications, in particular, database and spreadsheet applications. Because the format had been around for awhile before any attempt was made at standardization, not all CSV files use exactly the same format. Fortunately, Python's `csv` module does a good job of handling and hiding these differences so the programmer generally doesn't have to worry about them.

Microsoft Excel is perhaps the most common application used for making CSV files. Here is a sample CSV file (`working-with-data/data/population-by-state.csv`) in Microsoft Excel showing the United States population breakdown[12] over several years:

	A	B	C	D	E	F
1	State	2010	2011	2012	2013	201
2	Alabama	4,785,437	4,799,069	4,815,588	4,830,081	4,841,79
3	Alaska	713,910	722,128	730,443	737,068	736,28
4	Arizona	6,407,172	6,472,643	6,554,978	6,632,764	6,730,41
5	Arkansas	2,921,964	2,940,667	2,952,164	2,959,400	2,967,39
6	California	37,319,502	37,638,369	37,948,800	38,260,787	38,596,97

Open the CSV file in a text editor and you'll see this (lines are cut off):

12. The population data is from `https://www.census.gov/data/tables/time-series/demo/popest/2010s-state-total.html`. The 2020 numbers are extrapolated.

```
(.venv) …/working-with-data/Demos> python csv_reader.py
State,2010,2011,2012,2013,2014,2015,2016,2017,2018,2019,2020
Alabama,"4,785,437","4,799,069","4,815,588","4,830,081","4,841,79…
Alaska,"713,910","722,128","730,443","737,068","736,283","737,498…
Arizona,"6,407,172","6,472,643","6,554,978","6,632,764","6,730,41…
Arkansas,"2,921,964","2,940,667","2,952,164","2,959,400","2,967,3…
California,"37,319,502","37,638,369","37,948,800","38,260,787","3…
…
```

Python's `csv` module is used for:

1. Reading from a CSV file.
2. Creating a new CSV file.
3. Writing to an existing CSV file.

❖ 3.7.1. Reading from a CSV File

To read data from a CSV file:

1. Open the file using the built-in `open()` function with `newline` set to an empty string.
2. Pass the file object to the `csv.reader()` method.
3. Read the file row by row. Each row is a list of strings.

Demo 3.8: working-with-data/Demos/csv_reader.py

```
1.   import csv
2.
3.   csv_file = '../data/population-by-state.csv'
4.   with open(csv_file, newline='', encoding="utf-8") as csvfile:
5.       pops = csv.reader(csvfile)
6.       for row in pops:
7.           print(', '.join(row))
```

This will output the following (lines are cut off):

```
(.venv) …/working-with-data/Demos> python csv_dictreader_1.pyState, 2010, 2011, 2012,
2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020
Alabama, 4,785,437, 4,799,069, 4,815,588, 4,830,081, 4,841,799, 4,85…
Alaska, 713,910, 722,128, 730,443, 737,068, 736,283, 737,498, 741,45…
Arizona, 6,407,172, 6,472,643, 6,554,978, 6,632,764, 6,730,413, 6,82…
Arkansas, 2,921,964, 2,940,667, 2,952,164, 2,959,400, 2,967,392, 2,9…
California, 37,319,502, 37,638,369, 37,948,800, 38,260,787, 38,596,9…
…
```

DictReader

When retrieving rows using the `reader()` method, it's possible to manipulate them item by item, but you have to know the positions of the different fields. For a CSV with a lot of columns, that can be pretty difficult. You will likely find it easier to use a `DictReader`, which gives you access to the fields by key.

For example, in the population CSV, the first column holds the name of the state, and each subsequent columns holds the population for a given year. The following report shows the growth between 2010 and 2020 for each state.

Demo 3.9: working-with-data/Demos/csv_dictreader_1.py

```
1.   import csv
2.
3.   csvfile = '../data/population-by-state.csv'
4.   with open(csvfile, newline='', encoding="utf-8") as csvfile:
5.       pops = csv.DictReader(csvfile)
6.       print('Headers:', pops.fieldnames)
7.       for row in pops:
8.           # Convert to integers
9.           pop_2020 = int(row['2020'].replace(',', ''))
10.          pop_2010 = int(row['2010'].replace(',', ''))
11.
12.          growth = pop_2020 - pop_2010
13.
14.          # Use "," format spec to separate 1000s with commas
15.          print(f"{row['State']}: {pop_2020:,} - {pop_2010:,} = {growth:,}")
```

This will output the following:

```
Headers: ['State', '2010', '2011', '2012', '2013', '2014', '2015', '2016', '2017', '2018',
 '2019', '2020']
Alabama: 4,914,833 - 4,785,437 = 129,396
Alaska: 742,866 - 713,910 = 28,956
Arizona: 7,341,977 - 6,407,172 = 934,805
Arkansas: 3,030,315 - 2,921,964 = 108,351
California: 40,037,709 - 37,319,502 = 2,718,207
…
```

Things to notice:

1. `csv.DictReader` objects have a `fieldnames` property that contains a list holding the keys taken from the first row of data.
2. We use the `replace()` method of strings to get rid of the commas in the population numbers and then we `int()` the result.
3. When we print the result, we add the commas back in using formatting.

fieldnames

By default the `fieldnames` property of `csv.DictReader` objects contains a list holding the keys taken from the first row of data. If the CSV doesn't have a header row, you can pass `fieldnames` in when creating the `DictReader`:

```
fieldnames = ['State', '2010', '2011', '2012', '2013', '2014',
              '2015', '2016', '2017', '2018', '2019', '2020']
pops = csv.DictReader(csvfile, fieldnames)
```

Getting CSV Data as a List

Review the following Python code:

Demo 3.10: working-with-data/Demos/csv_dictreader_2.py

```
1.   import csv
2.
3.   def get_data_from_csv(csvfile):
4.       with open(csvfile, newline='', encoding="utf-8") as csvfile:
5.           data = csv.DictReader(csvfile)
6.           return data
7.
8.   def main():
9.       data = get_data_from_csv('../data/population-by-state.csv')
10.
11.      for row in data:
12.          print(row['State'])
13.
14.  main()
```

What do you expect to happen? Clearly, the intention is to print all the states in the "State" column of the CSV, but instead the code will error:

```
ValueError: I/O operation on closed file.
```


The reason it errors is that files opened using `with` are automatically closed at the end of the `with` block. An effective way of dealing with this issue is to convert the `csv.DictReader` object to a list and return that instead:

Demo 3.11: working-with-data/Demos/csv_dictreader_3.py

```
1.   import csv
2.
3.   def get_data_as_list_from_csv(csvfile):
4.       with open(csvfile, newline='', encoding="utf-8") as csvfile:
5.           data = csv.DictReader(csvfile)
6.           return list(data)
7.
8.   def main():
9.       data = get_data_as_list_from_csv('../data/population-by-state.csv')
10.
11.      for row in data:
12.          print(row['State'])
13.
14.  main()
```

Exercise 9: Finding Data in a CSV File

20 to 30 minutes

In this exercise, you will use your knowledge of Python lists and dictionaries to search data in a CSV file.

1. Create a new file and save it as `csv_search.py` in `working-with-data/Exercises`.
2. Write code that prompts the user for a state name and a year between 2010 and 2020 and then returns the population of that state in that year. The program should work like this:


```
…/working-with-data/Exercises> python csv_search.py
State: New York
Year: 2020
New York's population in 2020: 19,588,068.
```

3. The code in `working-with-data/Demos/csv_dictreader_3.py` should serve as a good reference / starting point.

Solution: working-with-data/Solutions/csv_search.py

```
1.  import csv
2.
3.  def get_data_as_list_from_csv(csvfile):
4.      with open(csvfile, newline='', encoding="utf-8") as csvfile:
5.          data = csv.DictReader(csvfile)
6.          return list(data)
7.
8.  def get_population(data, state, year):
9.      # Loop through data
10.     for row in data:
11.         # Is this the row that matches the passed-in state?
12.         if row['State'] == state:
13.             # Return the value for the column for the passed in year
14.             return row[year]
15.     return None # No matching state found
16.
17. def main():
18.     data = get_data_as_list_from_csv('../data/population-by-state.csv')
19.     state = input('State name: ')
20.     year = input('Year between 2010 and 2020: ')
21.     population = get_population(data, state, year)
22.     if population:
23.         print(f'{state}\'s population in {year}: {population}.')
24.     else:
25.         print(f'No state found matching "{state}".')
26.
27. main()
```


3.8. Creating a New CSV File

❖ 3.8.1. Writer

To write data to a CSV file using a `Writer` object:

1. Open the file using the built-in `open()` function in writing mode and with `newline` set to an empty string:

```
with open('../csvs/mlb-weight-over-time.csv', 'w', newline='') as csvfile:
```

2. Pass the file object to the `csv.writer()` method:

```
writer = csv.writer(csvfile)
```

3. Write the file row by row with the `writerow(sequence)` method or all at once with the `writerows(sequence_of_sequences)` method.

Note that the sequences mapping to rows may only contain strings and numbers.

The following example shows how you could write data retrieved from a database into a CSV file:

Demo 3.12: working-with-data/Demos/csv_writer.py

```
1.  import mysql.connector
2.  import csv
3.
4.  connection = mysql.connector.connect(
5.      host='lahman.csw1rmup8ri6.us-east-1.rds.amazonaws.com',
6.      user='python',
7.      passwd='python',
8.      db='lahmansbaseballdb'
9.  )
10.
11. query = """SELECT year(debut) AS year, avg(weight) AS weight
12. FROM people
13. WHERE debut is NOT NULL
14. GROUP BY year(debut)
15. ORDER BY year(debut)"""
16.
17. cursor = connection.cursor()
18. cursor.execute(query)
19. results = cursor.fetchall()
20.
21. cursor.close()
22. connection.close()
23.
24. csv_file = '../data/mlb-weight-over-time.csv'
25. with open(csv_file, 'w', newline='', encoding='utf-8') as csvfile:
26.     writer = csv.writer(csvfile)
27.     writer.writerow(['Year', 'Weight'])
28.     writer.writerows(results)
```

❖ 3.8.2. DictWriter

`csv.DictWriter()` works like `csv.writer()` except that instead of mapping lists onto rows, it maps dictionaries onto rows. As such, it requires a second argument: `fieldnames`, which it uses to determine the order in which the dictionary values get mapped. To illustrate this, let's look at a very simple example:

Demo 3.13: working-with-data/Demos/csv_dictwriter_1.py

```
1.   import csv
2.
3.   grades = [
4.       {
5.           "English": 97,
6.           "Math": 93,
7.           "Art": 74,
8.           "Music": 86
9.       },
10.      {
11.          "English": 89,
12.          "Math": 83,
13.          "Art": 97,
14.          "Music": 94
15.      }
16.  ]
17.
18.  csv_file = '../data/grades.csv'
19.  with open(csv_file, 'w', newline='', encoding='utf-8') as csvfile:
20.      fieldnames = ['Math', 'Art', 'English', 'Music']
21.      writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
22.      writer.writeheader()
23.      writer.writerows(grades)
```


This will create `grades.csv` with the following contents:

```
Math,Art,English,Music
93,74,97,86
83,97,89,94
```

Notice that the order of the fields maps to the `fieldnames` list.

To get the order of the fields from the first dictionary in the `grades` list, use:

```
fieldnames = grades[0].keys()
```

Try making that change in `csv_dictwriter_1.py`. After making that change and running the file again, `grades.csv` will hold the following contents:

```
English,Math,Art,Music
97,93,74,86
89,83,97,94
```

Appending to a CSV File

To add lines to an existing CSV file, just open in it in append mode:

Demo 3.14: working-with-data/Demos/csv_dictwriter_2.py

```
1.   import csv
2.
3.   grades = [
4.       {
5.           "English": 88,
6.           "Math": 88,
7.           "Art": 88,
8.           "Music": 88
9.       },
10.      {
11.          "English": 77,
12.          "Math": 77,
13.          "Art": 77,
14.          "Music": 77
15.      }
16.  ]
17.
18.  csv_file = '../data/grades.csv'
19.  with open(csv_file, 'a', newline='', encoding='utf-8') as csvfile:
20.      fieldnames = grades[0].keys()
21.      writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
22.      writer.writerows(grades)
```


When appending, you don't need to call `writeheader()`.

Exercise 10: Creating a CSV with DictWriter

10 to 15 minutes

1. Open `working-with-data/Exercises/csv_dictwriter.py` in your editor. It contains the same code we saw in `working-with-data/Demos/csv_writer.py`
2. Modify the code so that it uses a `DictWriter` instead of a `Writer` object.

Solution: working-with-data/Solutions/csv_dictwriter.py

```
1.  import mysql.connector
2.  import csv
3.
4.  connection = mysql.connector.connect(
5.      host='lahman.csw1rmup8ri6.us-east-1.rds.amazonaws.com',
6.      user='python',
7.      passwd='python',
8.      db='lahmansbaseballdb'
9.  )
10.
11. query = """SELECT year(debut) AS year, avg(weight) AS weight
12. FROM people
13. WHERE debut is NOT NULL
14. GROUP BY year(debut)
15. ORDER BY year(debut)"""
16.
17. cursor = connection.cursor(dictionary=True)
18. cursor.execute(query)
19. results = cursor.fetchall()
20.
21. cursor.close()
22. connection.close()
23.
24. csv_file = '../data/mlb-weight-over-time.csv'
25. with open(csv_file, 'w', newline='', encoding='utf-8') as csvfile:
26.     fieldnames = results[0].keys()
27.     writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
28.     writer.writeheader()
29.     writer.writerows(results)
```


3.9. Getting Data from the Web

❖ 3.9.1. The Requests Package

Python has a built-in `urllib` module for making HTTP requests, but the `Requests`[13] package is far more developer-friendly. In this section, you'll learn how the `Requests` package works and how to combine it with the `Beautiful Soup`[14] library to parse the code.

13. https://requests.readthedocs.io/en/master/
14. https://www.crummy.com/software/BeautifulSoup/bs4/doc/

The first thing is to install the `Requests` package. With your virtual environment activated, run:

```
pip install requests
```

Although all HTTP request methods (e.g., `post`, `put`, `head`,…) can be used, in most cases, you will use the `get` method using `requests.get()`, to which you will pass in the URL as a string like this:

Demo 3.15: working-with-data/Demos/using_requests.py

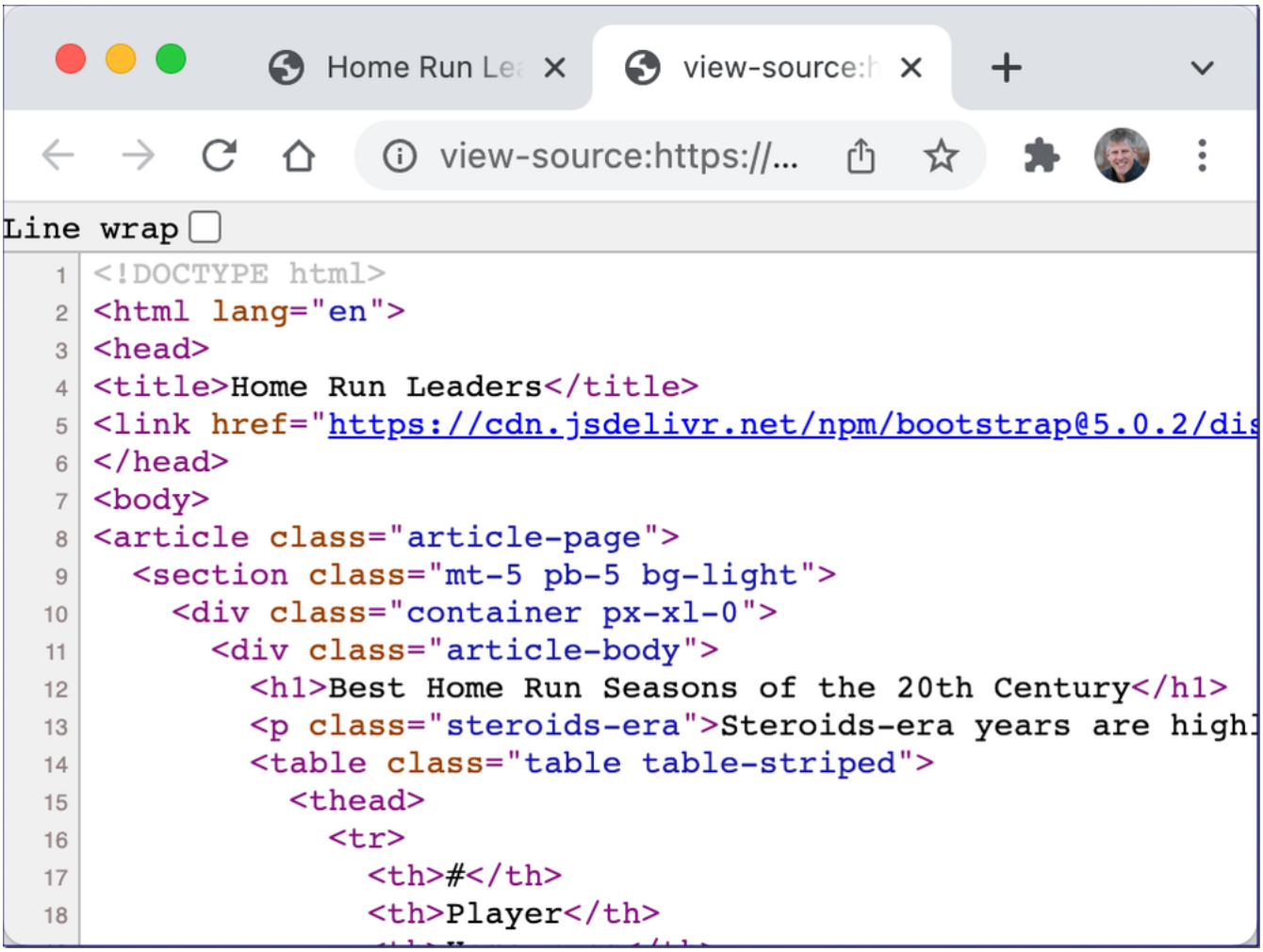
```
1.   import requests
2.
3.   url = 'https://static.webucator.com/media/public/documents/hrleaders.html'
4.   r = requests.get(url)
5.   content = r.text
6.   print(content[:125]) # print first 125 characters
```


This will grab all the content from the web page at `https://static.webucator.com/media/public/documents/hrleaders.html` and print out the first 125 characters. The result will be something like:

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>Home Run Leaders</title>
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.
```

To see where this code is coming from…

1. Navigate to `https://static.webucator.com/media/public/documents/hrleaders.html` in your web browser.
2. Right-click the page and select **View page source** or **View source**. You should see something like:



This is the code returned by the web server that the browser uses to draw the web page. Notice that the page begins with the text that our script output.

Custom Headers

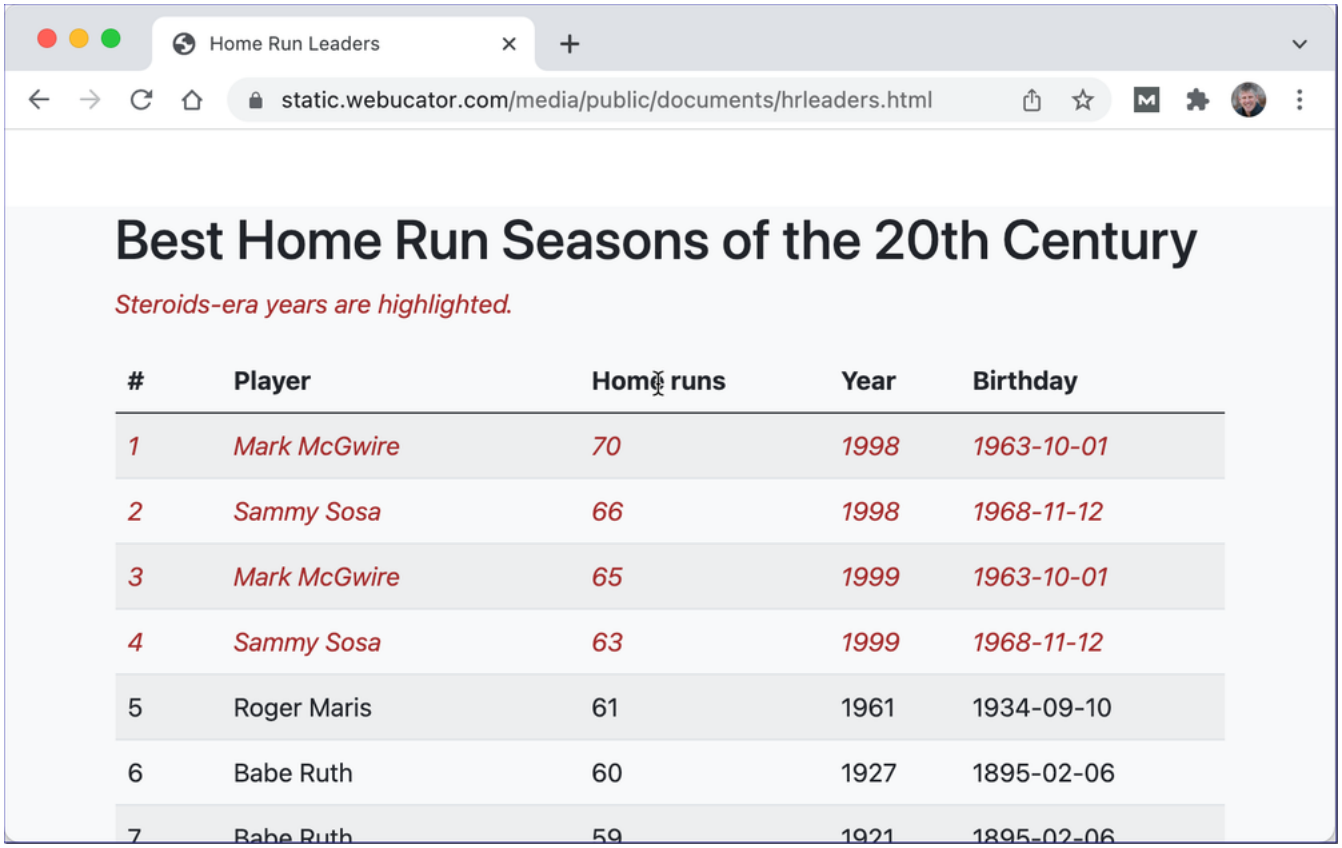
A web server might choose to block any requests that don't identify the user agent. You may be able to get around this by passing in headers that include a value for "user-agent", like this:

```
r = requests.get(url, headers={'user-agent': 'my-app/0.0.1'})
```

The `text` attribute of the `response` object returned by `requests.get()` will hold the full code of the HTML page delivered from the specified URL. In this case the URL is:

`https://static.webucator.com/media/public/documents/hrleaders.html`.

When viewed in a browser, the page looks like this:



❖ 3.9.2. HTML

Data in web pages is stored in HTML, a relatively simple markup language. Elements of an HTML page are marked up with tags. For example, a paragraph is started with an opening `<p>` tag and ended with a closing `</p>` tag. The page shown in the previous screenshot displays home-run records in a table. Each record is in a table row that is marked up like this:

```
<tr class="steroids-era">
  <td>1</td>
  <td>Mark McGwire</td>
  <td>70</td>
  <td>1998</td>
  <td>1963-10-01</td>
</tr>
```

- **T**able **r**ows begin with an opening `<tr>` tag and end with a closing `</tr>` tag.
- Within table rows, **t**able **d**ata cells begin with an opening `<td>` tag and end with a closing `</td>` tag, and **t**able **h**eader cells begin with an opening `<th>` tag and end with a closing `</th>` tag.

Tags often have *attributes* for further defining the element. Attributes usually come in name-value pairs.

Note that attributes only appear in the opening tag, like so:

```
<tagname att1="value" att2="value">Element content</tagname>
```

Some of the home-run table rows, like the one just shown, have a `class` attribute with the value of "steroids-era":

```
<tr class="steroids-era">
```


It is common to search for elements by tag name and by class.

Writing code to extract data using string manipulation and regular expressions would be quite challenging. `Beautiful Soup` to the rescue!

❖ 3.9.3. Beautiful Soup

Beautiful Soup is a Python library for extracting HTML and XML data. It is often used to find specific content on a web page, a process known as "scraping."

Install Beautiful Soup in your virtual environment:

```
pip install beautifulsoup4
```

You can choose between several options[15] for parsing the HTML content. BeautifulSoup recommends `lxml`. Install that as well:

15. https://www.crummy.com/software/BeautifulSoup/bs4/doc/#specifying-the-parser-to-use

lxml and Case

The `lxml` parser converts all elements and attributes to lowercase letters, so you should only search for lowercase tags and attributes.

```
pip install lxml
```

To get a feel for how Beautiful Soup works, start up Python in the terminal:

1. The first step is to import Beautiful Soup:

   ```
   >>> from bs4 import BeautifulSoup
   ```

2. Import requests and get some data to work with:

   ```
   >>> import requests
   >>> url = 'https://static.webucator.com/media/public/documents/hrleaders.html'
   >>> r = requests.get(url)
   >>> content = r.text
   ```


3. Using the content of that web page, create a `BeautifulSoup` object using the `lxml` parser:

   ```
   >>> soup = BeautifulSoup(content, 'lxml')
   ```

4. Use the `find()`, `find_all()`, or `select()` methods to find the tags you are looking for.

   - `find()` returns a single `bs4.element.Tag` object, which we will just call a `Tag` object.
   - `find_all()` and `select()` each return a `bs4.element.ResultSet`, which is essentially a list of `Tag` objects.

   The `find()` and `find_all()` methods provide a lot of options for finding tags. We'll use `find_all()` to illustrate, but `find()` works in the same way.

5. Get all the table rows (`tr` elements) in the content:

   ```
   >>> trs = soup.find_all('tr')
   >>> len(trs)
   101
   ```

6. Because `find_all()` is the most common method used, there is a shortcut method for using it. You can simply treat `soup` as a method itself, like this:

```
>>> trs = soup('tr')
```

Find all the table data cells:

```
>>> tds = soup('td')
>>> len(tds)
500
```

7. BeautifulSoup `Tag` objects can also take the `find()`, `find_all()`, and `select()` methods. Find all the `td` elements in the first table row:

```
>>> first_row = trs[0]
>>> first_row.find_all('td')
[]
```

There are none, because the first row contains table header elements (`<th>` tags). Just as we can treat the main `soup` object as a method as a shortcut for `find_all()`, we can treat a `Tag` object as a method as well:


```
>>> first_row('th')
[<th>#</th>, <th>Player</th>, <th>Home runs</th>, <th>Year</th>, <th>Birth  ↵↵
day</th>]
```

8. As we mentioned earlier, it is common to search for elements by their tag name and class value. You can pass a value for the class as the second argument of `find_all()` and `find()`. The following code would find the first table row with the "steroids-era" class:

```
>>> soup.find('tr', 'steroids-era')
<tr class="steroids-era">
<td>1</td>
<td>Mark McGwire</td>
<td>70</td>
<td>1998</td>
<td>1963-10-01</td>
</tr>
```

9. You can also pass in a `text` argument to search for elements that contain certain text:

```
>>> ruths = soup.find_all('td', text='Babe Ruth')
>>> len(ruths)
9
```

10. Use the `parent` property to get a `Tag`'s parent `Tag`:

```
>>> first_ruth_row = ruths[0].parent
>>> first_ruth_row
<tr>
<td>6</td>
<td>Babe Ruth</td>
<td>60</td>
<td>1927</td>
<td>1895-02-06</td>
</tr>
```

11. In addition to searching on the text of an element, you can access the text of `Tag` object using the `text` property:


```
>>> ruth_birth_day = first_ruth_row.find_all('td')[-1].text
>>> ruth_birth_day
'1895-02-06'
```

Other Attributes

In addition to finding elements by name and class value, you can find elements by any of their attributes. To do so, pass in name-value pairs when you create the `soup` object. For example, the following code would get all `a` elements (HTML links) with a `target` attribute set to `_blank`:

```
soup.find_all('a', target='_blank')
```

The following file prompts the user for a URL and then searches the web page for links with a `target` of "_blank":

Demo 3.16: working-with-data/Demos/a_blanks.py

```
1.   import requests
2.   from bs4 import BeautifulSoup
3.
4.
5.   def get_content():
6.       url = input('Enter a URL: ')
7.       r = requests.get(url)
8.       return r.text
9.
10.  def main():
11.      content = get_content()
12.      soup = BeautifulSoup(content, 'lxml')
13.
14.      external_links = soup.find_all('a', target='_blank')
15.
16.      found = False
17.      for i, link in enumerate(external_links, 1):
18.          found = True
19.          print(f'{i}. {link}')
20.
21.      if not found:
22.          print('None found.')
23.
24.  main()
```


Run the file and enter the URL of your choice to see if it has any links that target "_blank".

At the time of this writing, `https://www.stackoverflow.com` had some such links.

Finding Non-Section-508-Compliant Images

Images on HTML pages are created with the `<img>` tag. People with impaired vision rely on the value of the `img` element's `alt` attribute to know what the image represents. All `img` elements on web pages should include `alt` attributes with values. You can check whether an attribute is included or not using a boolean value:

```
soup.find_all('img', alt=False)
```

The following file prompts the user for a URL and then searches the web page for `img` elements with no `alt` value:

Demo 3.17: working-with-data/Demos/imgs_wo_alt.py

```
1.   import requests
2.   from bs4 import BeautifulSoup
3.
4.
5.   def get_content():
6.       url = input('Enter a URL: ')
7.       r = requests.get(url)
8.       return r.text
9.
10.  def main():
11.      content = get_content()
12.      soup = BeautifulSoup(content, 'lxml')
13.
14.      images = soup.find_all('img', alt=False)
15.
16.      found = False
17.      for i, img in enumerate(images, 1):
18.          found = True
19.          print(f'{i}. {img}')
20.
21.      if not found:
22.          print('None found.')
23.
24.  main()
```


Run the file and enter the URL of your choice to see if it has any images are missing the `alt` value.

At the time of this writing, `https://www.syracuse.com` had some such images.

Exercise 11: HTML Scraping

30 to 45 minutes

In this exercise, you will try to scrape data from the web page at:

`https://static.webucator.com/media/public/documents/hrleaders.html`

Working in the terminal or in a file, try to find:

1. The last cell of the last row of the table.
2. The number of records that occurred during the steroids era.
3. The name of the player with the most home runs in a single season. Note that the first `tr` element contains the column headings in `th` tags. That `tr` is contained within a `thead` element. The rows with the data in them are all contained within a single `tbody` element. You want to get the text in the second cell of the first row in the `tbody` element. The relevant HTML segment looks like this:


```
<tbody>
<tr class="steroids-era">
  <td>1</td>
  <td>Mark McGwire</td>
  <td>70</td>
  <td>1998</td>
  <td>1963-10-01</td>
</tr>…
```

4. The most home runs Willie Mays ever got in a season.
5. The list of player names who hit 50 or more home runs in a single season. Do this one in a Python file. It should print out the player names. There will be 17, beginning with:

```
1. Mark McGwire
2. Sammy Sosa
3. Roger Maris
```

Solution

First, get the BeautifulSoup object:

```
>>> import requests
>>> from bs4 import BeautifulSoup
>>> url = 'https://static.webucator.com/media/public/documents/hrleaders.html'

>>> r = requests.get(url)
>>> content = r.text
>>> soup = BeautifulSoup(content, 'lxml')
```

1. The last cell of the last row of the table:

```
>>> trs = soup.find_all('tr')
>>> trs[-1]('td')[-1]
<td>1936-08-08</td>
```

2. The number of records that occurred during the steroids era:

```
>>> len(soup.find_all('tr', 'steroids-era'))
29
```


3. The name of the player with the most home runs in a single season:

```
>>> data_container = soup.find('tbody')
>>> first_row = data_container.find('tr')
>>> player_name = first_row.find('td').text
>>> player_name = first_row.find_all('td')[1].text
>>> player_name
'Mark McGwire'
```

4. The most home runs Willie Mays ever got in a season:

```
>>> mays_first_td = soup.find('td', text='Willie Mays')
>>> mays_hr_record = mays_first_td.parent.find_all('td')[2].text
>>> mays_hr_record
'52'
```

5. The number of players who hit 50 or more home runs in a single season: See the following file:

Solution: working-with-data/Solutions/fifty_hrs.py

```
1.  import requests
2.  from bs4 import BeautifulSoup
3.
4.  # Constants to express which content is in which cells
5.  NUM = 0
6.  PLAYER = 1
7.  HRS = 2
8.  YEAR = 3
9.  BIRTH_DAY = 4
10.
11. def get_content():
12.     url = 'https://static.webucator.com/media/public/documents/hrleaders.html'
13.     r = requests.get(url)
14.     return r.text
15.
16. def get_soup(content):
17.     return BeautifulSoup(content, 'lxml')
18.
19. def get_players(soup):
20.     data_container = soup.find('tbody')
21.
22.     # Get all table rows in the tbody
23.     rows = data_container.find_all('tr')
24.
25.     players = []
26.     # Loop through the rows
27.     for row in rows:
28.         # Get int value of HRS text
29.         hrs = int(row.find_all('td')[HRS].text)
30.         if hrs >= 50:
31.             player = row.find_all('td')[PLAYER].text
32.             # Add the name of the player to players
33.             #  but only if they're not already in there
34.             if player not in players:
35.                 players.append(player)
36.         if hrs < 50:
37.             return players # No need to keep looking
38.
39.     return players # Just in case all have 50 or more hrs
40.
41. def main():
42.     content = get_content()
43.     soup = get_soup(content)
44.     players = get_players(soup)
```

```
45.
46.      for i, player in enumerate(players, 1):
47.          print(f'{i}. {player}')
48.
49.  main()
```

3.10. XML

XML (e**X**tensible **M**arkup **L**anguage) is a meta-language; that is, it is a language in which other languages are created. In XML, data is "marked up" with tags, similar to HTML tags. In fact, one version of HTML, called XHTML, is an XML-based language, which means that XHTML follows the syntax rules of XML.

XML is used to store data or information; this data might be intended to be read by people or by machines. It can be highly structured data such as data typically stored in databases or spreadsheets, or loosely structured data, such as data stored in letters or manuals.

Beautiful Soup is also used to parse XML documents. `lxml` is the recommended parser for parsing XML as well as HTML:

```
soup = BeautifulSoup(content, 'lxml')
```

The home-runs page we have been working with has an XML version at:

`https://static.webucator.com/media/public/documents/hrleaders.xml`

The XML for the page contains `record` elements, an abridged version of which is shown here:

```
<records>
  <record>
    <Player>Mark McGwire</Player>
    <HR>70</HR>
    <Year>1998</Year>
    <Birthday>1963-10-01</Birthday>
  </record>
  <record>
    <Player>Sammy Sosa</Player>
    <HR>66</HR>
    <Year>1998</Year>
    <Birthday>1968-11-12</Birthday>
  </record>
  …
</records>
```


Note that each player name is stored in the `Player` tag, but `lxml` will convert all tags and attributes to lowercase letters, so we will search for "player". Here is the code to list all players in the XML file:

Demo 3.18: working-with-data/Demos/soup_xml.py

```
1.   import requests
2.   from bs4 import BeautifulSoup
3.
4.   url = 'https://static.webucator.com/media/public/documents/hrleaders.xml'
5.   r = requests.get(url)
6.   content = r.text
7.
8.   soup = BeautifulSoup(content, 'lxml')
9.
10.  players = soup.find_all('player')
11.  for i, player in enumerate(players, 1):
12.      print(f'{i}. {player.text}')
```

3.11. JSON

JSON stands for **J**ava**S**cript **O**bject **N**otation. According to the official JSON website[16], JSON is:

1. A lightweight data-interchange format.
2. Easy for humans to read and write.
3. Easy for machines to parse and generate.

Numbers 1 and 3 are certainly true. Number 2 depends on the type of human. Experienced Python programmers will find the JSON syntax extremely familiar as it is uses the same syntax as Python's `dict` and `list` objects.

Here is an example of JSON holding weather data:

16. https://json.org

Demo 3.19: working-with-data/data/weather.json

```
{
  "list": [{
    "dt": 1581886800,
    "main": {
      "temp": 36.86,
      "feels_like": 29.97,
      "temp_min": 33.22,
      "temp_max": 36.86,
      "humidity": 98
    },
    "wind": {
      "speed": 7.11,
      "deg": 243
    },
    "dt_txt": "2020-02-16 21:00:00"
  }, {
    "dt": 1581897600,
    "main": {
      "temp": 30.65,
      "feels_like": 23.94,
      "temp_min": 27.93,
      "temp_max": 30.65,
      "humidity": 94
    },
    "wind": {
      "speed": 4.85,
      "deg": 254
    },
    "dt_txt": "2020-02-17 00:00:00"
  }],
  "city": {
    "id": 5140405,
    "name": "Syracuse",
    "country": "US",
    "population": 145170,
    "timezone": -18000,
    "sunrise": 1581854499,
    "sunset": 1581892559
  }
}
```

This is a simplified version of the JSON returned from the OpenWeatherMap API's current weather data[17]. As you can see, it is formatted just like a Python `dict`:

1. The value for the "list" key is a `list` of `dict`s, each representing the weather for the subsequent three-hour period.
2. The value for the "city" key is a `dict` containing data about the city.

After assigning the `dict` to a `weather` variable, you could get the name of the city and the max temparature for the next period as follows:

```
city_name = weather['city']['name']
max_temp = weather['list'][0]['main']['temp_max']
```

Many organizations, including Google, Twitter, Facebook, Reddit, and Microsoft, provide APIs that return data in JSON.[18] Each API has its own rules and parameters. Most, including the OpenWeatherMap API, require an API key.

Getting an OpenWeatherMap API Key

The OpenWeatherMap API has a free tier, but it requires an API key. To get an API key, sign up at `https://home.openweathermap.org/users/sign_up`.

To get the forecast for Syracuse, NY in JSON, go to this URL (replacing **yourapikey** with a valid API key):

```
https://api.openweathermap.org/data/2.5/forecast/?q=Syracuse,New+York,us&units=imperi ↵↵
al&APPID=yourapikey
```

Notice the parameters passed in the URL:

1. **APPID** – a valid API key.
2. **q**: a string in the format:

   ```
   city_name,state_name,country_code
   ```

17. https://api.openweathermap.org/data/2.5/forecast/daily
18. For a list of many JSON APIs, see `https://www.programmableweb.com/category/all/apis?data_format=21173`.

3. **units**: the system of measurement (`imperial` for fahrenheit, `metric` for celsius)

The following code uses the API to report the forecasted max temperature in Syracuse, NY for the next ten days:

Demo 3.20: working-with-data/Demos/weather.py

```
1.  import requests
2.  from datetime import datetime
3.
4.  API_KEY = 'abca198b092b0295697beb48914a442c'
5.  FEED = 'https://api.openweathermap.org/data/2.5/forecast/'
6.
7.
8.  def main():
9.      city = 'Syracuse'
10.     state = 'New York'
11.     country_code = 'us'
12.
13.     params = {
14.         'q': city + ',' + state + ',' + country_code,
15.         'units': 'imperial',
16.         'APPID': API_KEY
17.     }
18.
19.     r = requests.get(FEED, params)
20.     print(r.url) # prints the URL created using the params
21.
22.     weather = r.json()
23.
24.     fmt_in = '%Y-%m-%d %H:%M:%S'
25.     fmt_out = '%A, %B %d, %Y at %I %p'
26.     for item in weather['list']:
27.         max_temp = item['main']['temp_max']
28.         dt = item['dt_txt']
29.         day_time = datetime.strptime(dt, fmt_in).strftime(fmt_out)
30.         print(f'High on {day_time} in {city}: {max_temp} fahrenheit.')
31.
32. main()
```


The `json()` method of the response from `requests.get()` converts the JSON code to a Python `dict`. From there, you can work with the Python `dict` object as you normally would.

Exercise 12: JSON Home Runs

20 to 30 minutes

In this exercise, you will work with the JSON data at the following URL:

`https://static.webucator.com/media/public/documents/hrleaders.json`

1. Open a new file and save it as `hrs_json.py` in `working-with-data/Exercises`.
2. Using the data returned from `https://static.webucator.com/media/public/documents/hrleaders.json`, write code to print out the number of home runs each player in the JSON data hit. The first ten results are shown here:


```
1. Mark McGwire hit 70 home runs in 1998.
2. Sammy Sosa hit 66 home runs in 1998.
3. Mark McGwire hit 65 home runs in 1999.
4. Sammy Sosa hit 63 home runs in 1999.
5. Roger Maris hit 61 home runs in 1961.
6. Babe Ruth hit 60 home runs in 1927.
7. Babe Ruth hit 59 home runs in 1921.
8. Jimmie Foxx hit 58 home runs in 1932.
9. Hank Greenberg hit 58 home runs in 1938.
10. Hack Wilson hit 56 home runs in 1930.
```

Solution: working-with-data/Solutions/hrs_json.py

```
1.    import requests
2.
3.    data="https://static.webucator.com/media/public/documents/hrleaders.json"
4.
5.    r = requests.get(data)
6.    records = r.json()
7.
8.    for i, record in enumerate(records, 1):
9.        player = record['Player']
10.       hrs = record['HR']
11.       year = record['Year']
12.       print(f'{i}. {player} hit {hrs} home runs in {year}.')
```


Don't forget to deactivate your virtual environment:

```
deactivate
```

Conclusion

In this lesson, you have learned to work with data stored in databases, CSV files, HTML, XML, and JSON.

LESSON 4
Testing and Debugging

Topics Covered

☑ Testing performance with timers and the `timeit` module.

☑ The `unittest` module.

Introduction

In this lesson, you will learn to test the performance and the functionality of your Python code.

4.1. Testing for Performance

❖ 4.1.1. `time.perf_counter()`

The `time` module includes a `perf_counter()` method that is used to precisely measure relative times. The following code illustrates:

Demo 4.1: testing-debugging/Demos/time_diff.py

```
1.   import time
2.
3.   t1 = time.perf_counter()
4.   t2 = time.perf_counter()
5.   print(t1, t2, t2-t1)
```

Note that the value returned by `time.perf_counter()` is only meaningful when compared to other values returned by `time.perf_counter()` at different times. If we run the same file multiple times, `t1` and `t2` will have different values, but the time difference between them should be similar:

```
…/testing-debugging/Demos> python time_diff.py
0.0178728 0.0178734 5.999999999999062e-07
…/testing-debugging/Demos> python time_diff.py
0.0167957 0.0167963 5.999999999999062e-07
…/testing-debugging/Demos> python time_diff.py
```

Scientific Notation

5.999999999999062e-07 seconds is a very small amount of time. The number is displayed in scientific notation and is the equivalent of 0.000000599999999999062. That's approximately .00006 milliseconds. If you need to brush up on your scientific notation, check out the short video at `https://youtu.be/JrbvjFqFITI`.

The following code shows how to use `time.perf_counter()` to compare how fast two pieces of code run:

Demo 4.2: testing-debugging/Demos/compare_code_speed1.py


```
1.  import random
2.  import time
3.
4.  start_time = time.perf_counter()
5.  numbers = str(random.randint(1, 100))
6.  for i in range(1000):
7.      num = random.randint(1, 100)
8.      numbers += ',' + str(num)
9.  end_time = time.perf_counter()
10. td1 = end_time - start_time
11.
12. start_time = time.perf_counter()
13. numbers = [str(random.randint(1, 100)) for i in range(1000)]
14. numbers = ', '.join(numbers)
15. end_time = time.perf_counter()
16. td2 = end_time - start_time
17.
18. print(f"""Time Delta 1: {td1}
19. Time Delta 2: {td2}""")
```

Both snippets of code create strings from 1,000 random numbers between 1 and 100.

1. The first snippet (lines 5-8) uses the `+=` operator to repeatedly append to the `numbers` string.

2. The second snippet (lines 13-14) uses a list comprehension to create a list of random numbers and then uses the `join()` method to convert the list to a string.

We capture the time before and after each code snippet and then print the results. Here are the results from running it four times:

```
…/testing-debugging/Demos> python time_diff.py
…/testing-debugging/Demos> python compare_code_speed1.py
Time Delta 1: 0.0011696999999999992
Time Delta 2: 0.0009586000000000004
…/testing-debugging/Demos> python compare_code_speed1.py
Time Delta 1: 0.0011660000000000004
Time Delta 2: 0.0009369000000000009
…/testing-debugging/Demos> python compare_code_speed1.py
Time Delta 1: 0.0011210999999999999
Time Delta 2: 0.0009372999999999986
…/testing-debugging/Demos> python compare_code_speed1.py
Time Delta 1: 0.0011487999999999984
Time Delta 2: 0.0009316000000000012
```


To get an ever more accurate comparison of the efficiency of the two methods, we can run each snippet through a loop 1,000 times and then compare the results:

Demo 4.3: testing-debugging/Demos/compare_code_speed2.py

```
1.   import random
2.   import time
3.
4.   start_time = time.perf_counter()
5.   for j in range(1000):
6.       numbers = str(random.randint(1, 100))
7.       for i in range(1000):
8.           num = random.randint(1, 100)
9.           numbers += ',' + str(num)
10.  end_time = time.perf_counter()
11.  td1 = end_time - start_time
12.
13.  start_time = time.perf_counter()
14.  for j in range(1000):
15.      numbers = [str(random.randint(1, 100)) for i in range(1000)]
16.      numbers = ', '.join(numbers)
17.  end_time = time.perf_counter()
18.  td2 = end_time - start_time
19.
20.  print(f"""Time Delta 1: {td1}
21.  Time Delta 2: {td2}""")
```


Here is the output of this file:

```
…/testing-debugging/Demos> python compare_code_speed2.py
Time Delta 1: 1.2513799
Time Delta 2: 0.8971576000000001
```

❖ 4.1.2. The `timeit` Module

Python comes with a built-in `timeit` module that does everything we just saw (and more) for you. The two methods you will use most often are:

1. `timeit.timeit()`
2. `timeit.repeat()`

`timeit.timeit()`

The `timeit()` method can take multiple parameters, but the most important are:

1. `stmt` – The statement to run.
2. `setup` – Any code that should be run before the time testing begins. The time it takes to execute this code will **not** be included in the results.
3. `number` – The number of times to run it. This argument is optional, but it defaults to 1000000, which could take an awful long time to run.

It returns the number of seconds it took to run `stmt` `number` times.

Often, you will have to rewrite a piece of code to test it using `timeit`. One way of doing this is to create temporary functions to hold the different pieces of code you want to compare. Here's an example:

Demo 4.4: testing-debugging/Demos/using_timeit1.py

```
1.  import random
2.  from timeit import timeit
3.
4.  def string_nums1():
5.      numbers = str(random.randint(1, 100))
6.      for i in range(1000):
7.          num = random.randint(1, 100)
8.          numbers += ', ' + str(num)
9.
10. def string_nums2():
11.     numbers = [str(random.randint(1, 100)) for i in range(1000)]
12.     numbers = ', '.join(numbers)
13.
14. td1 = timeit(string_nums1, number=1000)
15. td2 = timeit(string_nums2, number=1000)
16.
17. print("Results from using timeit()")
18. print(td1, td2, sep="\n")
19. print('-' * 70)
20.
21. print('string_nums1 compared to string_nums2:')
22. print(f'{td1/td2:.2%}')
```


This will show that the `string_nums1()` function is about 30% less efficient than the `string_nums2()` function:

```
…/testing-debugging/Demos> python using_timeit1.py
Results from using timeit()
1.1584021
0.8860760999999997
------------------------------------------------------------------------
string_nums1 compared to string_nums2:
130.73%
```

Note that, because the two `string_nums` functions are both defined in the *global namespace*, they have access to the imported `random` module, so we do not need to import that via the `setup` parameter.

Namespaces

In Python, a *namespace* is a *names-to-objects mapping*. Namespaces are closely related to scope and are used to distinguish between identically named attributes and variables defined in different modules. The namespace of the top-level of the running module is *globals*. And the namespace of each imported module is the name of that module. Functions defined within a module have their own *local* namespaces.

Another way to do this is to place the code you want to test in strings and then import `random` in the `setup` argument, like this:

Demo 4.5: testing-debugging/Demos/using_timeit2.py

```
1.   from timeit import timeit
2.
3.   str_nums1 = """
4.   numbers = str(random.randint(1, 100))
5.   for i in range(1000):
6.       num = random.randint(1, 100)
7.       numbers += ', ' + str(num)"""
8.
9.   str_nums2 = """
10.  numbers = [str(random.randint(1, 100)) for i in range(1000)]
11.  numbers = ', '.join(numbers)"""
12.
13.  td1 = timeit(str_nums1, number=1000, setup='import random')
14.  td2 = timeit(str_nums2, number=1000, setup='import random')
     -------Lines 15 through 21 Omitted-------
```

In this code, the code in the strings is executed within `timeit`'s namespace, so we need to use `setup` to import `random` into that namespace.

A third way, as of Python 3.5, is to specify the `globals` namespace, which will give the code run by `timeit()` access to global attributes and imported modules:

Demo 4.6: testing-debugging/Demos/using_timeit3.py

```
1.   import random
2.   from timeit import timeit
3.
4.   str_nums1 = """
5.   numbers = str(random.randint(1, 100))
6.   for i in range(1000):
7.       num = random.randint(1, 100)
8.       numbers += ', ' + str(num)"""
9.
10.  str_nums2 = """
11.  numbers = [str(random.randint(1, 100)) for i in range(1000)]
12.  numbers = ', '.join(numbers)"""
13.
14.  td1 = timeit(str_nums1, number=1000, globals=globals())
15.  td2 = timeit(str_nums2, number=1000, globals=globals())
     -------Lines 16 through 22 Omitted-------
```


All of these methods are fine and should yield similar results.

`timeit.repeat()`

The `repeat()` method is similar to the `timeit()` method, but it runs the loop multiple times and returns a list with the results of each repetition. In addition to `stmt`, `setup`, and `number`, the `repeat()` method has a `repeat` parameter, which takes the number of times to repeat the loop. The default value of `repeat` is `5`.[19]

Here is the previous example using `repeat()` instead of `timeit()`:

19. In Python 3.6 and earlier, the default value for `repeat` was `3`.

Demo 4.7: testing-debugging/Demos/using_repeat.py

```
1.   import random
2.   from timeit import repeat
3.
4.   str_nums1 = """
5.   numbers = str(random.randint(1, 100))
6.   for i in range(1000):
7.       num = random.randint(1, 100)
8.       numbers += ', ' + str(num)"""
9.
10.  str_nums2 = """
11.  numbers = [str(random.randint(1, 100)) for i in range(1000)]
12.  numbers = ', '.join(numbers)"""
13.
14.  tds1 = repeat(str_nums1, number=1000, repeat=4, globals=globals())
15.  tds2 = repeat(str_nums2, number=1000, repeat=4, globals=globals())
16.
17.  print("Results from using repeat()")
18.  print(tds1, tds2, sep="\n")
19.  print('-' * 70)
20.
21.  print('str_nums1 compared to str_nums2:')
22.  print(f'{sum(tds1)/sum(tds2):.2%}')
```


This will output something like this:

```
…/testing-debugging/Demos> python using_repeat.py
Results from using repeat()
[1.0138493, 1.302279, 1.1477145000000002, 1.2157394999999998]
[0.8648758000000001, 0.8432685000000006, 0.8369980000000004, 0.8368100000000007]
----------------------------------------------------------------------
str_nums1 compared to str_nums2:
138.37%
```

After four iterations of looping through each approach 1,000 times, it's pretty clear that the second approach is faster.

Using `timeit` Interactively

You may find that using `timeit()` interactively is more convenient for small snippets of code.

Let's try the code that generates a list of 1,000 random integers in the range of 1 to 100:

```
>>> import random
>>> from timeit import timeit
>>> timeit(', '.join([str(random.randint(1, 100)) for i in range(1000)]))
0.008025599999999855
```


Your output will likely be slightly different from the output shown above.

Exercise 13: Comparing Times to Execute

10 to 15 minutes

1. Open `testing-debugging/Exercises/compare_code.py` in your editor.
2. Notice that the functions use random words rather than receiving a word as a parameter as they normally would.
3. Write code to compare the time it takes to run the different functions.

Exercise Code 13.1: testing-debugging/Exercises/compare_code.py

```
1.   import re
2.   import random
3.
4.   def get_word():
5.       words = ['Charlie', 'Woodstock', 'Snoopy', 'Lucy', 'Linus',
6.                'Schroeder', 'Patty', 'Sally', 'Marcie']
7.       return random.choice(words).upper()
8.
9.   def green_glass_door_1():
10.      word = get_word()
11.      prev_letter = ''
12.      for letter in word:
13.          letter = letter.upper()
14.          if letter == prev_letter:
15.              return True
16.          prev_letter = letter
17.      return False
18.
19.  def green_glass_door_2():
20.      word = get_word()
21.      pattern = re.compile(r'(.)\1')
22.      return pattern.search(word)
```

Solution: testing-debugging/Solutions/compare_code.py

```
1.  import re
2.  import random
3.  from timeit import repeat
4.
5.  def get_word():
6.      words = ['Charlie', 'Woodstock', 'Snoopy', 'Lucy', 'Linus',
7.               'Schroeder', 'Patty', 'Sally', 'Marcie']
8.      return random.choice(words).upper()
9.
10. def green_glass_door_1():
11.     word = get_word()
12.     prev_letter = ''
13.     for letter in word:
14.         letter = letter.upper()
15.         if letter == prev_letter:
16.             return True
17.         prev_letter = letter
18.     return False
19.
20. def green_glass_door_2():
21.     word = get_word()
22.     pattern = re.compile(r'(.)\1')
23.     return pattern.search(word)
24.
25. tds1 = repeat(green_glass_door_1, number=1000, repeat=4)
26. tds2 = repeat(green_glass_door_2, number=1000, repeat=4)
27.
28. print(tds1, tds2, sep="\n")
29. print('-' * 70)
30.
31. print('green_glass_door_1 compared to green_glass_door_2:')
32. print('{:.2%}'.format(sum(tds1)/sum(tds2)))
```


Here are our results:

```
[0.0012243000000000046, 0.0011920000000000056, 0.0011614999999999959, 0.0011576]
[0.0016824000000000006, 0.0017012, 0.0013999999999999985, 0.0017819999999999989]
----------------------------------------------------------------------
green_glass_door_1 compared to green_glass_door_2:
72.12%
```

It appears that the regular expression version is less efficient than the other.

When Efficiency Matters

You don't need to test all your code for efficiency. A little bit faster is not always a little bit better, especially if it comes at the cost of code clarity. We recommend that you follow this procedure:

1. Get your code working.
2. Clean up your code to make it as readable as possible. This includes adding comments.
3. **If** your code has an efficiency problem, use `timeit` to identify the source of the problem and fix it.

This is not to say that you shouldn't be concerned with efficiency while coding. If you already know one method is significantly more efficient than another, all else being equal, use the more efficient method. Just don't let the quest for efficiency slow you down.

4.2. The `unittest` Module

The built-in `unittest` module provides a framework for writing unit tests by extending the `unittest.TestCase` class. It is best understood through an example.

We'll start with three functions, the third of which has an error:

Demo 4.8: testing-debugging/Demos/functions_error.py

```
1.    def prepend(s, c):
2.        return c + s
3.
4.    def append(s, c):
5.        return s + c
6.
7.    def insert(s, c, pos):
8.        return s[0:pos] + c + s[pos:-1] # wrong
```

Classes

In this lesson, we will be discussing *classes*. You will learn to create your own classes in the **Classes and Objects** lesson (see page 153). In that lesson, you will write tests for the classes you create. So, we have a *catch 22*: it is helpful to understand classes when learning to write tests, but it is also helpful to understand testing when learning to write classes. Fortunately, you only need to know a little about classes for the rest of this lesson:

1. Classes are created using the `class` keyword.
2. Classes are named in upper camel case (e.g., `TestCase`).
3. Classes generally have methods, which are functions tied to the class.
4. Classes refer to themselves (or rather to the objects they create) as `self`, and they call their own methods with `self.method_name()`.

Don't get hung up on the class syntax for now. Look to understand the methods (functions) within the class.

To test our functions using the `unittest` module, we must do the following (at a minimum):

1. Import the `unittest` module.
2. Create a class that extends `unittest.TestCase`.
3. Write test methods. Note that the names of the test methods must start with the string "test".

To run the tests, call `unittest.main()`, but we only want to do that when we run the test file directly, so will put it in the following conditional (see page 48):

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

Here is a class to test the three functions above along with code to create and run the suite of tests:

Demo 4.9: testing-debugging/Demos/unit_test_functions.py

```
1.   import unittest
2.   from functions_error import prepend, append, insert
3.
4.   class TestMyMethods(unittest.TestCase):
5.       def test_prepend(self):
6.           self.assertEqual(prepend('bar', 'foo'), 'foobar')
7.       def test_append(self):
8.           self.assertEqual(append('bar', 'foo'), 'barfoo')
9.       def test_insert(self):
10.          self.assertEqual(insert('wetor', 'buca', 2), 'webucator')
11.
12.  if __name__ == '__main__':
13.      unittest.main()
```

When we run the test suite, we get the following result:

```
…/testing-debugging/Demos> python unit_test_functions.py
.F.
======================================================================
FAIL: test_insert (__main__.TestMyMethods)
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".\unit_test_functions.py", line 10, in test_insert
    self.assertEqual(insert('wetor', 'buca', 2), 'webucator')
AssertionError: 'webucato' != 'webucator'
- webucato
+ webucator
?         +


----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=1)
```


This shows that three tests ran and one failed. And it gives details on the test that failed. Our code expected `insert('wetor', 'buca', 2)` to equal "webucator", but instead it resulted in "webucato". That means there is something wrong with our `insert()` function. **Can you fix it?**

❖ 4.2.1. Unittest Test Files

You can run many test files at once from the command line with the following command:

```
python -m unittest discover directory_with_tests
```

That will search the `directory_with_tests` folder for Python files with names that begin with "test". For example, in the `testing-debugging/Demos` folder, there are two files with helper functions in them:

1. `math_functions.py`
2. `string_functions.py`

Each of these has an associated unit test file:

1. `test_math_functions.py`
2. `test_string_functions.py`

All four files are shown here:

Demo 4.10: testing-debugging/Demos/math_functions.py

```
1.  import math
2.
3.  def round_down(f):
4.      return int(f) # doesn't work for negative numbers
5.
6.  def round_up(f):
7.      return math.ceil(f)
```

Note that the `int()` function does not round down. It strips the decimal portion of the number, leaving just the integer. For negative numbers, this effectively rounds up (e.g., `-5.4` becomes `-5`). We should have used `math.floor()` instead.

Demo 4.11: testing-debugging/Demos/string_functions.py

```
1.   import random
2.   import string
3.   import re
4.
5.   def prepend(s, c):
6.       return c + s
7.
8.   def append(s, c):
9.       return s + c
10.
11.  def insert(s, c, pos):
12.      return s[0:pos] + c + s[pos:-1] # wrong
```

Recall that `s[first_pos:last_pos]` returns a slice that starts with the character at `first_pos` and includes all the characters up to *but not including* the character at `last_pos`. In `insert()`, we should have used `s[pos:]` to get a slice that includes the last character of the original string.

Demo 4.12: testing-debugging/Demos/test_math_functions.py


```
1.   import unittest
2.   from math_functions import *
3.
4.   class TestMathFunctions(unittest.TestCase):
5.
6.       def test_round_down(self):
7.           self.assertEqual(round_down(1.3), 1)
8.           self.assertEqual(round_down(-1.3), -2)
9.           self.assertEqual(round_down(1.7), 1)
10.          self.assertEqual(round_down(-1.7), -2)
11.          self.assertEqual(round_down(0), 0)
12.
13.      def test_round_up(self):
14.          self.assertEqual(round_up(1.3), 2)
15.          self.assertEqual(round_up(-1.3), -1)
16.          self.assertEqual(round_up(1.7), 2)
17.          self.assertEqual(round_up(-1.7), -1)
18.          self.assertEqual(round_up(0), 0)
19.
20.  if __name__ == '__main__':
21.      unittest.main()
```

Notice that this file imports all the functions from the `math_functions` module.

Demo 4.13: testing-debugging/Demos/test_string_functions.py

```
1.   import unittest
2.   from string_functions import *
3.
4.   class TestStringFunctions(unittest.TestCase):
5.
6.       def test_prepend(self):
7.           self.assertEqual(prepend('bar', 'foo'), 'foobar')
8.
9.       def test_append(self):
10.          self.assertEqual(append('bar', 'foo'), 'barfoo')
11.
12.      def test_insert(self):
13.          self.assertEqual(insert('wetor', 'buca', 2), 'webucator')
14.
15.  if __name__ == '__main__':
16.      unittest.main()
```


Notice that this file imports all the functions from the `string_functions` module.

To discover and run all the unit tests in the `testing-debugging/Demos` folder, open `testing-debugging` in the terminal and run:

```
python -m unittest discover Demos
```

Or, if you want to see a list of all the tests that run, include the `-v` option, like this:

```
python -m unittest discover Demos -v
```

Here are the results with the `-v` option included (run in Windows PowerShell):

```
…/Python/testing-debugging> python -m unittest discover Demos -v
test_round_down (test_math_functions.TestMathFunctions) ... FAIL
test_round_up (test_math_functions.TestMathFunctions) ... ok
test_append (test_string_functions.TestStringFunctions) ... ok
test_insert (test_string_functions.TestStringFunctions) ... FAIL
test_prepend (test_string_functions.TestStringFunctions) ... ok

======================================================================
FAIL: test_round_down (test_math_functions.TestMathFunctions)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\ndunn\OneDrive\Documents\Webucator\Courseware\complete-courses\PYT-
111_print\ClassFiles\testing-debugging\Demos\test_math_functions.py", line 8, in
test_round_down
    self.assertEqual(round_down(-1.3), -2)
AssertionError: -1 != -2

======================================================================
FAIL: test_insert (test_string_functions.TestStringFunctions)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\ndunn\OneDrive\Documents\Webucator\Courseware\complete-courses\PYT-
111_print\ClassFiles\testing-debugging\Demos\test_string_functions.py", line 13, in
test_insert
    self.assertEqual(insert('wetor', 'buca', 2), 'webucator')
AssertionError: 'webucato' != 'webucator'
- webucato
+ webucator
?         +

----------------------------------------------------------------------
Ran 5 tests in 0.007s

FAILED (failures=2)
```

This shows that five tests were run and two failed.

Exercise 14: Fixing Functions

10 to 15 minutes

In this exercise, you will correct the functions that failed the unit tests in our demos. The same files we used in the demos are in the `testing-debugging/Exercises` folder.

1. At the terminal, run the command to discover and run unit tests in the `testing-debugging/Exercises` folder.
2. Open `testing-debugging/Exercises/math_functions.py`.
    A. Fix the function that failed the unittest.
3. Open `testing-debugging/Exercises/string_functions.py`.
    A. Fix the function that failed the unittest.
4. Run the command to discover and run unit tests again.
5. If any unit tests failed, go back and fix the associated functions.

Challenge

Try writing your own simple function and an associated unit test.

Solution: testing-debugging/Solutions/string_functions.py

```
1.    import random
2.    import string
3.    import re
4.
5.    def prepend(s,c):
6.        return c + s
7.
8.    def append(s,c):
9.        return s + c
10.
11.   def insert(s,c,pos):
12.       return s[0:pos] + c + s[pos:]
```

Solution: testing-debugging/Solutions/math_functions.py

```
1.    import math
2.
3.    def round_down(f):
4.        return math.floor(f)
5.
6.    def round_up(f):
7.        return math.ceil(f)
```


4.3. Special `unittest.TestCase` Methods

The `unittest.TestCase` class includes special `setUp()` and `tearDown()` methods that run before and after each test. You can use them to:

1. Instantiate (and clean up) class instances to use in tests.
2. Open and close files, database connections, network connections, etc.
3. Start and/or stop any server processes needed to run the tests.

These methods (and others like them) create the working environment for the tests. This working environment is called a *fixture*. To see how these methods work:

1. Open `testing-debugging/Demos/beatles/beatles.py` in your editor and review the code.

2. Open `testing-debugging/Demos/beatles/test_beatles.py` in your editor. Note that the `TestBeatles` class contains two test methods:
    A. `test_select()` – checks to make sure the `select()` method returns a list.
    B. `test_select_one()` – checks to make sure the `select()` method returns a tuple.
3. The `TestBeatles` class also contains `setUp()` and `tearDown()` methods, which run before and after each test.
    A. The `setUp()` method instantiates a `beatles` object and runs the `create()` and `insert()` methods to create and populate the `beatles` table. It also prints "Setting up".
    B. The `tearDown()` method runs the `close()` method to close the cursor and connection. It also prints "Tearing down".
4. Run the `test_beatles.py` file. It should output the following:

```
Setting up
Tearing down
.Setting up
Tearing down
.
----------------------------------------------------------------------
Ran 2 tests in 0.003s

OK
```


You wouldn't usually print "Setting up" and "Tearing down". We do this only to show that the fixture methods get called once for each test.

❖ 4.3.1. Assert Methods

The `assertEqual()` method we used in our examples is the most common, but there are many other assert methods available in `TestCase` classes, including:

- `assertNotEqual()`
- `assertTrue()`
- `assertFalse()`
- `assertIs()`

- `assertIsNot()`
- `assertIsNone()`
- `assertIsNotNone()`
- `assertIn()`
- `assertNotIn()`
- `assertIsInstance()`
- `assertNotIsInstance()`
- `assertRaises()`

See `https://docs.python.org/3/library/unittest.html#assert-methods` for the full list.

❖ 4.3.2. Additional Setup Methods

- The `setUpClass()` and `tearDownClass()` methods are used to run code before any of the tests in a `unittest.TestCase` class begins and after they all end. These must be implemented as class methods.
- The `setUpModule()` and `tearDownModule()` methods are used to run code at the beginning and end of the module containing test cases. These are module-level functions that are not part of any `unittest.TestCase` class. They are usually defined at the top of the module containing one or more `unittest.TestCase` classes.

See `https://docs.python.org/3/library/unittest.html` for full documentation on the unit testing framework.

Conclusion

In this lesson, you have learned to test the performance of different pieces of code and to create unit tests to test your Python code.

LESSON 5
Classes and Objects

Topics Covered

☑ Classes and objects in Python.

☑ Instance methods, class methods, and static methods.

☑ Properties.

☑ Decorators.

☑ Subclasses and inheritance.

Introduction

An object is something that has attributes and/or behaviors, meaning it *is* certain ways and *does* certain things. In the real world, everything could be considered an object. Some objects are tangible, like rocks, trees, tennis racquets, and tennis players. And some objects are intangible, like words, colors, tennis swings, and tennis matches. In this lesson, you will learn how to write object-oriented Python code.

5.1. Attributes

If you can say "x is y" or "x has y," then `x` is an object, and `y` is an attribute of `x`. Some examples:

1. *The rock that he is holding* is *heavy*. Heavy is an attribute of the specific rock he is holding. More generally, rocks have weight.
2. *The apple tree in our backyard* has *four branches*. The four branches are attributes of that specific tree. More generally, trees have branches.
3. *Venus Williams' swing* is *strong*. Strong is an attribute of Venus Williams' swing. More generally, tennis swings have a strength.
4. *The final match* had three *sets*. The three sets are attributes of the specific match. More generally, matches have sets.

5. *Serena Williams* has a *first-serve percentage of 57.2%*. A 57.2% first-serve percentage is an attribute of Serena Williams. More generally, tennis players have a first-serve percentage.

Attributes are generally nouns (e.g., branches) or adjectives (e.g., heavy). In Python, we could write the statements above like this:

```
rock_he_holds.weight = 'heavy'
backyard_apple_tree.branches = [branch1, branch2, branch3, branch4]
venus.swing = 'strong'
final_match.sets = [set1, set2, set3]
serena.serve1 = .572
```

5.2. Behaviors

If you can say "x does" then `does` is a behavior of `x`. Some examples of behaviors:

1. *The rock falls* fast. Rocks can fall.
2. *The apple tree in our back yard* first *bore fruit* on August 23, 2002. Trees can bear fruit.
3. Venus Williams' *swing hit* the ball. Tennis swings can hit things.
4. The final *match ended* at 11:45 in the morning. Matches can end.
5. *Serena Williams served*. Tennis players can serve.

Behaviors are verbs and behaviors of objects are called *methods*, which are simply functions defined within a class definition. In Python, we could write the statements above like this:

```
rock_he_holds.fall('fast')
backyard_apple_tree.bear_fruit(datetime.date(2002, 8, 23))
venus.swing.hit(ball)
final_match.end(datetime.time(11, 45))
serena.serve()
```

5.3. Classes vs. Objects

A class is a template for an object. An object is an *instance* of a class. When we say Serena Williams is a tennis player, we are saying that Serena Williams is an object of the `TennisPlayer` class. There are

other tennis players who have the same attributes and behaviors as Serena Williams, but not in the same way. For example, Serena has a winning percentage. Her sister Venus also has a winning percentage. So do Roger Federer and Rafael Nadal. But their winning percentages are all different. They also all have backhands, but they don't all have the same backhand. Roger Federer has a one-handed backhand, while the others all have two-handed backhands. If you needed to express that in code, you could do it this way:

```
serena.two_handed_backhand = True
venus.two_handed_backhand = True
roger.two_handed_backhand = False
rafa.two_handed_backhand = True
```

In programming, we use classes to define what attributes and behaviors an object has or can have. In Python, we do this with the `class` keyword, like this:

```
class Player:
    pass
```


We then create an instance of that class like this:

```
serena = Player()
```

❖ 5.3.1. Everything Is an Object

Before we go further with this, let's take a look at some of the classes and objects we have already worked with in Python. In Python, everything is an object. Strings are objects, lists are objects, integers are objects, functions are objects. Even classes themselves are objects. Python includes two built-in functions that help identify the class of an object:

- `type(obj)` – Returns the object's type, which is essentially a synonym for class.
- `isinstance(obj, class_type)` – Returns `True` if `obj` is an instance of `class_type`. Otherwise, returns `False`.

Take a look at the following examples:

```
>>> type('Hello')
<class 'str'>
>>> isinstance('Hello', str)
True
>>> type(1)
<class 'int'>
>>> isinstance(1, int)
True
>>> type(['a','b','c'])
<class 'list'>
>>> isinstance(['a','b','c'], list)
True
>>> type((1,2,3))
<class 'tuple'>
>>> isinstance((1,2,3), tuple)
True
>>> type({'a':1, 'b':2})
<class 'dict'>
>>> isinstance({'a':1, 'b':2}, dict)
True
```


The `print()` function is an instance of the `builtin_function_or_method`:

```
>>> type(print)
<class 'builtin_function_or_method'>
```

There are some built-in functions that are really classes and not functions. Examples are `tuple`, `list`, `dict`, and `range`. The following code shows that `age` is of type `range` and `range` is of type `type` (meaning that it is a class):

```
>>> age = range(0, 100)
>>> type(age)
<class 'range'>
>>> type(range)
<class 'type'>
```

Again, *type* is just a synonym for *class*.[20]

20. Historically, there were some differences, but those differences are largely academic.

❖ 5.3.2. Creating Custom Classes

Now, let's return to our `Player` class:

```
class Player:
    pass
```

By convention, classes are named using upper camel case (e.g., `MyClass`).[21] Check out the following:

```
>>> class Player:
...     pass
...
>>> serena = Player()
>>> type(serena)
<class '__main__.Player'>
>>> type(Player)
<class 'type'>
>>> isinstance(serena, Player)
True
```


This shows:

1. That the `Player` class is of type `type`.
2. That the instance of `Player` is of type `__main__.Player`. The `__main__` tells us that the class was defined at the top level. Don't worry about that for now.
3. That `serena` is an instance of `Player`.

5.4. Attributes and Methods

Let's see how to create a class that creates dice objects:

Demo 5.1: classes-objects/Demos/Die1.py

```
1.  class Die:
2.      pass
```

21. The `tuple`, `list`, `dict`, and `range` classes are exceptions because they are usually used like functions rather than classes.

We could import this class and instantiate a new `Die` object like this:

```
import Die1

die = Die1.Die()
```

Or we could import the `Die` class directly:

```
from Die1 import Die

die = Die()
```

Try this yourself by opening a Python prompt at `classes-objects/Demos` and running the code:

```
>>> import Die1
>>> die = Die1.Die()
>>> die
<Die1.Die object at 0x00000212F7800AF0>
>>> from Die1 import Die
>>> die = Die()
>>> die
<Die1.Die object at 0x00000212F79DAB20>
```


This `Die` class doesn't currently define any attributes or methods, so it's not very useful. The appropriate attributes and methods would depend on what type of application we want to build. No matter what the application is though, we are going to want to include an *initialization method*. An initialization method is a special method in Python, `__init__()`, that is automatically called when an object is instantiated (i.e., created). The purpose of the `__init__()` method is to set the initial attributes of the new object.

Double Underscores

Note that the double underscores surrounding `init` indicate that this is a special method (a *magic method*) in Python. You should never name your own methods in this way.

For now, let's initialize `Die` objects with a single attribute: `sides` with a default value of `6`:

Demo 5.2: classes-objects/Demos/Die2.py

```
1.   class Die:
2.       def __init__(self, sides=6):
3.           self.sides = sides
```

The first parameter of every standard method defined in a class, including the `__init__()` method, is `self`,[22] which is a reference to the object being created. Methods can have any number of additional parameters. Our `__init__()` method takes one additional parameter: `sides`, which it assigns to `self.sides`. When we create a `Die` object, we do not pass in anything for `self`; the object itself gets passed in automatically. We just pass in a value for `sides`. The result is that our new `Die` object has a `sides` attribute, which holds an integer.

Forgetting Your self

If you are new to creating your own classes, it may take you a while to get used to including `self` as the first parameter of your methods. If you forget to include it, you will get an error similar to the following when you try to call that method on your class instance:

```
xyz_method() takes 0 positional arguments but 1 was given
```

Look at the class definition for `Die` again and notice that in the `__init__()` method we assign `sides` to `self.sides`. Remember that `self` is the object created by the class. The value we pass in for `sides` is assigned to the `sides` attribute of `self` as the following test code demonstrates:

22. You could name the first parameter by a different name, but you really shouldn't. It would just be self-punishment.

Demo 5.3: classes-objects/Demos/test_Die2.py

```
1.   import unittest
2.   from Die2 import Die
3.
4.   class TestDieFunctions(unittest.TestCase):
5.
6.       def setUp(self):
7.           self.die1 = Die()
8.           self.die2 = Die(8)
9.
10.      def test_init(self):
11.          self.assertEqual(self.die1.sides, 6)
12.          self.assertEqual(self.die2.sides, 8)
13.
14.  if __name__ == '__main__':
15.      unittest.main()
```


Run this test file to see how it works:

```
…/classes-objects/Demos> python test_Die2.py -v
test_init (__main__.TestDieFunctions) ... ok

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

OK
```

Both of our asserts passed, meaning that `self.sides` contains what we expect it to.

Derived Attribute Values

It might not always be as straightforward as directly assigning a passed-in argument to an attribute. Consider this class definition for `Circle`:

Demo 5.4: classes-objects/Demos/Circle1.py

```
1.   import math
2.
3.   class Circle:
4.       def __init__(self, val, prop='r'):
5.           if prop == 'r': # radius
6.               self.radius = val
7.           elif prop == 'd': # diameter
8.               self.radius = val / 2
9.           elif prop == 'c': # circumference
10.              self.radius = val / (2 * math.pi)
11.          elif prop == 'a': # area
12.              self.radius = (val / math.pi) ** .5
13.          else:
14.              raise Exception('prop must be r, d, c, or a')
15.
16.          self.diameter = self.radius * 2
17.          self.circumference = self.radius * 2 * math.pi
18.          self.area = self.radius ** 2 * math.pi
```


In this case, a `Circle` object is initialized with two parameters: `val` and `prop`, neither of which is directly assigned to the object. Rather, we use the values passed in to determine four of the object's attributes:

1. `radius`
2. `diameter`
3. `circumference`
4. `area`

Let's test the class with the following test code:

Demo 5.5: classes-objects/Demos/test_Circle1.py

```
1.   import unittest
2.   import math
3.   from Circle1 import Circle
4.
5.   class TestCircleFunctions(unittest.TestCase):
6.
7.       def setUp(self):
8.           self.circle_r = Circle(5, 'r')
9.           self.circle_d = Circle(10, 'd')
10.          self.circle_c = Circle(10 * math.pi, 'c')
11.          self.circle_a = Circle(25 * math.pi, 'a')
12.
13.      # Test circle_r
14.      def test_circle_radius(self):
15.          self.assertEqual(self.circle_r.radius, 5)
16.          self.assertEqual(self.circle_r.diameter, 10)
17.          self.assertEqual(self.circle_r.circumference, 10 * math.pi)
18.          self.assertEqual(self.circle_r.area, 25 * math.pi)
19.
20.      # Test circle_d
21.      def test_circle_diameter(self):
22.          self.assertEqual(self.circle_d.radius, 5)
23.          self.assertEqual(self.circle_d.diameter, 10)
24.          self.assertEqual(self.circle_d.circumference, 10 * math.pi)
25.          self.assertEqual(self.circle_d.area, 25 * math.pi)
26.
27.      # Test circle_c
28.      def test_circle_circumference(self):
29.          self.assertEqual(self.circle_c.radius, 5)
30.          self.assertEqual(self.circle_c.diameter, 10)
31.          self.assertEqual(self.circle_c.circumference, 10 * math.pi)
32.          self.assertEqual(self.circle_c.area, 25 * math.pi)
33.
34.      # Test circle_a
35.      def test_circle_area(self):
36.          self.assertEqual(self.circle_a.radius, 5)
37.          self.assertEqual(self.circle_a.diameter, 10)
38.          self.assertEqual(self.circle_a.circumference, 10 * math.pi)
39.          self.assertEqual(self.circle_a.area, 25 * math.pi)
40.
41.  if __name__ == '__main__':
42.      unittest.main()
```

Running this file should return the following, indicating that the class calculates the attribute values correctly:

```
…/classes-objects/Demos> python test_Circle1.py -v
test_circle_area (__main__.TestCircleFunctions) ... ok
test_circle_circumference (__main__.TestCircleFunctions) ... ok
test_circle_diameter (__main__.TestCircleFunctions) ... ok
test_circle_radius (__main__.TestCircleFunctions) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

Let's now add a `resize_by()` method to our `Circle` class:

Demo 5.6: classes-objects/Demos/Circle2.py

```
1.   import math
2.
3.   class Circle:
4.       def __init__(self, val, prop='r'):
5.           if prop == 'r':
6.               self.radius = val
7.           elif prop == 'd':
8.               self.radius = val / 2
9.           elif prop == 'c':
10.              self.radius = val / (2 * math.pi)
11.          elif prop == 'a':
12.              self.radius = (val / math.pi) ** .5
13.          else:
14.              raise Exception('prop must be r, d, c, or a')
15.
16.          self.diameter = self.radius * 2
17.          self.circumference = self.radius * 2 * math.pi
18.          self.area = self.radius ** 2 * math.pi
19.
20.      def resize_by(self, amount):
21.          self.radius *= (1 + amount)
22.          self.diameter = self.radius * 2
23.          self.circumference = self.radius * 2 * math.pi
24.          self.area = self.radius ** 2 * math.pi
```

Again, we pass `self` into the method. We also pass in an `amount`. We use the value of `amount` to change `radius` and then we change the other attributes based on the new `radius`.

Let's test the class with the following test code:

Demo 5.7: classes-objects/Demos/test_Circle2.py

```
1.   import unittest
2.   import math
3.   from Circle2 import Circle
4.
5.   class TestCircleFunctions(unittest.TestCase):
6.
7.       def setUp(self):
     -------Lines 8 through 12 Omitted-------
13.          self.circle_resized = Circle(5, 'r')
14.          self.circle_resized.resize_by(.5) # grow by 50%
     -------Lines 15 through 39 Omitted-------
40.      def test_circle_resized(self):
41.          self.assertEqual(self.circle_resized.radius, 7.5)
42.          self.assertEqual(self.circle_resized.diameter, 15)
43.          self.assertEqual(self.circle_resized.circumference, 15 * math.pi)
44.          self.assertEqual(self.circle_resized.area, 7.5 * 7.5 * math.pi)
45.
46.  if __name__ == '__main__':
47.      unittest.main()
```


Running this file should return the following:

```
…/classes-objects/Demos> python test_Circle2.py -v
test_circle_area (__main__.TestCircleFunctions) ... ok
test_circle_circumference (__main__.TestCircleFunctions) ... ok
test_circle_diameter (__main__.TestCircleFunctions) ... ok
test_circle_radius (__main__.TestCircleFunctions) ... ok
test_circle_resized (__main__.TestCircleFunctions) ... ok

----------------------------------------------------------------------
Ran 5 tests in 0.000s

OK
```

Exercise 15: Adding a roll() Method to Die

15 to 25 minutes

Currently, we have a `Die` class to create `die` objects with some number of `sides`, but we have no way to roll the die. In this exercise, you will add a `roll()` method to the `Die` class.

1. Navigate to `classes-objects/Exercises` and open `Die.py` in your editor.
2. Add a `roll()` method that returns an integer between `1` and `sides`. You will need to import the `random` module.
3. Test your solution by creating an instance of `Die` and calling the `roll()` method several times.

Challenge

Write code to roll the die 100,000 times and then use a `Counter` object to create a list that shows how many times each side was rolled. It should output something like this:

```
[(1, 16422), (2, 16596), (3, 16567), (4, 16761), (5, 16951), (6, 16703)]
```

Solution: classes-objects/Solutions/Die1.py

```
1.    import random
2.
3.    class Die:
4.        def __init__(self, sides=6):
5.            self.sides = sides
6.
7.        def roll(self):
8.            roll = random.randint(1, self.sides)
9.            return roll
```

Solution: classes-objects/Solutions/roll_die1.py

```
1.    from Die1 import Die
2.
3.    die = Die()
4.
5.    roll = die.roll()
6.    print(roll)
```


Challenge Solution: classes-objects/Solutions/roll_die1_challenge.py

```
1.    from collections import Counter
2.    from Die1 import Die
3.
4.    die = Die()
5.
6.    rolls = []
7.    for i in range(100000):
8.        roll = die.roll()
9.        rolls.append(roll)
10.
11.   c = Counter(rolls)
12.   c_sorted = sorted(c.items())
13.
14.   print(c_sorted)
```

5.5. Private Attributes

Many object-oriented programming languages have the concept of *private attributes* – attributes of the instance objects that can only be modified within the class definition. To understand the need for private attributes, consider what would happen if we set a value for the radius of our circle `c` like this:

```
c.radius = 25
```


That would change the value of `radius`, but would not change the values of `diameter`, `circumference`, and `area`, and so, the circle's `radius` would now be out of sync with its other attributes. In languages that allow for private attributes, you would explicitly mark those attributes private and only allow access to them through getter and setter methods, like this:

Demo 5.8: classes-objects/Demos/Circle3.py

```
1.   import math
2.   class Circle:
3.       def __init__(self, val, prop='r'):
4.           if prop == 'r':
5.               self.set_radius(val)
6.           elif prop == 'd':
7.               self.set_diameter(val)
8.           elif prop == 'c':
9.               self.set_circumference(val)
10.          elif prop == 'a':
11.              self.set_area(val)
12.          else:
13.              raise Exception('prop must be r, d, c, or a')
14.
15.      def set_radius(self, r):
16.          self._radius = r
17.          self._diameter = r * 2
18.          self._circumference = r * 2 * math.pi
19.          self._area = r ** 2 * math.pi
20.
21.      def get_radius(self):
22.          return self._radius
23.
24.      def set_diameter(self, d):
25.          self.set_radius(d / 2)
26.
27.      def get_diameter(self):
28.          return self._diameter
29.
30.      def set_circumference(self, c):
31.          self.set_radius(c / (2 * math.pi))
32.
33.      def get_circumference(self):
34.          return self._circumference
35.
36.      def set_area(self, a):
37.          self.set_radius((a / math.pi) ** .5)
38.
39.      def get_area(self):
40.          return self._area
41.
42.      def resize_by(self, amount):
43.          r = self._radius * (1 + amount)
44.          self.set_radius(r)
```

Notice that the attributes are now preceded by an underscore (e.g., `_radius`). That is a convention to indicate that the attribute is meant to be private (i.e., not accessed directly from outside of the class definition). Instead, each attribute should be retrieved with its *getter* and set with its *setter*. The `set_radius()` setter sets all the "private" attributes and the other setters (e.g., `set_diameter()`) just hand off the work to `set_radius()`.

To access the attributes' values, we use the getters:

```
>>> from Circle3 import Circle
>>> c = Circle(10, 'd')
>>> a = c.get_area()
>>> a
78.53981633974483
>>> c.set_radius(8)
>>> a = c.get_area()
>>> a
201.06192982974676
```

Note that in Python there is no way to actually prevent developers from accessing the "private" attributes as the following example (continued from the preceding code) illustrates:


```
>>> c._radius = 5
>>> a = c.get_area()
>>> print(a)
201.06192982974676
```

The new area should have been set back to `78.53981633974483` to correspond to a radius of `5`, but notice that setting `c._radius` did not cause the `area` of `c` to get updated. Neither did it result in an error, so the developer won't know that something went wrong. Python developers just need to know not to mess with attributes that begin with underscores.

5.6. Properties

While using `get_` and `set_` methods like we did in our previous example works, it is not the Pythonic way of creating getters and setters. Python developers prefer being able to access attributes directly:

Not Pythonic

```
c.set_area(25)
a = c.get_area()
```

Pythonic

```
c.area = 25
a = c.area
```

This is handled in Python through *properties*. Think of a property as an attribute with a defined getter and possibly a setter and a deleter as well.

There are two ways to create properties: with the `property()` function and with the `@property` *decorator*.[23]

❖ 5.6.1. Creating Properties with the property() Function


```
class Circle:
    def __init__(self):
        self._radius = None

    def get_radius(self):
        return self._radius

    def set_radius(self, r):
        self._radius = r

    radius = property(get_radius, set_radius)
```

The code above makes use of `get_` and `set_` methods like many other object-oriented languages, but then it uses the built-in `property()` function to create a `radius` property, which allows `radius` to be got and set directly using `c.radius`.

23. Decorators are functions that add functionality to (i.e., "decorate") other functions.

❖ 5.6.2. Creating Properties using the @property Decorator

```
class Circle:
    def __init__(self):
        self._radius = None

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, r):
        self._radius = r
```


`@property` is a decorator, which turns the method directly following it into a getter method.

`@radius.setter` is also a decorator, which turns the method directly following it into a setter method for `radius`.

Here is our `Circle` class with properties defined:

Demo 5.9: classes-objects/Demos/Circle4.py

```
     -------Lines 1 through 15 Omitted-------
16.      def resize_by(self, amount):
17.          r = self._radius * (1 + amount)
18.          self.set_radius(r)
19.
20.      @property
21.      def radius(self):
22.          return self._radius
23.
24.      @radius.setter
25.      def radius(self, r):
26.          self._radius = r
27.          self._diameter = r * 2
28.          self._circumference = r * 2 * math.pi
29.          self._area = r ** 2 * math.pi
30.
31.      @property
32.      def diameter(self):
33.          return self._diameter
34.
35.      @diameter.setter
36.      def diameter(self, d):
37.          self.radius = d / 2
38.
39.      @property
40.      def circumference(self):
41.          return self._circumference
42.
43.      @circumference.setter
44.      def circumference(self, c):
45.          self.radius = c / (2 * math.pi)
46.
47.      @property
48.      def area(self):
49.          return self._area
50.
51.      @area.setter
52.      def area(self, a):
53.          self.radius = (a / math.pi) ** .5
```

Forgetting the Underscore

A common error when using the `@property` decorator is to forget the underscore when attempting to return the attribute, like this:

```
class Foo:
    @property
    def name(self):
        return self.name

    a = Foo()
    print(a.name)
```

This will cause an infinite loop and result in a "maximum recursion depth exceeded while calling a Python object" error, because the `return` statement is trying to return `self.name`, which directs Python back to the `name()` getter, which tries again to return `self.name`. This is bad. So, don't forget your underscores.

Exercise 16: Properties

15 to 25 minutes

In this exercise, you will convert `get_` methods to properties.

1. Open `classes-objects/Exercises/Simulation.py` in your editor.
2. Notice the `get_mean()`, `get_median()`, and `get_mode()` methods, which take advantage of the `statistics` module to calculate average results of a die rolled many times.
3. Run the code to see how it works. Here is a sample script you can run (using the `Die` class with the `roll` method assigned in an earlier exercise):

```
>>> from Die import Die
>>> from Simulation import Simulation
>>> die = Die()
>>> sim = Simulation(die.roll, 1000)
>>> sim.get_mean()
3.453
>>> sim.get_median()
3.0
>>> sim.get_mode()
1
```


4. Convert the three `get_` methods to properties and test your solution with the same code as above but replacing the `get_` methods statement with property references:

```
>>> from Die1 import Die
>>> from Simulation import Simulation
>>> die = Die()
>>> sim = Simulation(die.roll, 1000)
>>> sim.mean
3.453
>>> sim.median
3.0
>>> sim.mode
1
```

Solution: classes-objects/Solutions/Simulation.py

```
1.   import statistics as stats
2.
3.   class Simulation:
4.       def __init__(self, fnct_to_run, iterations):
5.           self._fnct_to_run = fnct_to_run
6.           self._iterations = iterations
7.           self._results = []
8.           self.run()
9.
10.      def run(self):
11.          for i in range(self._iterations):
12.              result = self._fnct_to_run()
13.              self._results.append(result)
14.
15.      @property
16.      def mean(self):
17.          return stats.mean(self._results)
18.
19.      @property
20.      def median(self):
21.          return stats.median(self._results)
22.
23.      @property
24.      def mode(self):
25.          try:
26.              return stats.mode(self._results)
27.          except:
28.              return None
```


5.7. Objects that Track their Own History

In the `Simulation` class from the last exercise, we keep track of the rolls, but a die could keep track of its own history as well:

Demo 5.10: classes-objects/Demos/Die3.py

```
1.   import random
2.
3.   class Die:
4.       def __init__(self, sides=6):
5.           if type(sides) != int or sides < 1:
6.               raise Exception('sides must be a positive integer.')
7.           self._sides = sides
8.           self._rolls = []
9.
10.      @property
11.      def rolls(self):
12.          return self._rolls
13.
14.      def roll(self):
15.          roll = random.randint(1, self._sides)
16.          self._rolls.append(roll)
17.          return roll
```

Each time the `roll()` method is called the `die` instance will automatically append the result of the roll to its `_rolls` property.

Demo 5.11: classes-objects/Demos/roll_die3.py

```
1.   from collections import Counter
2.   from Die3 import Die
3.
4.   die = Die()
5.
6.   for i in range(100000):
7.       roll = die.roll()
8.
9.   rolls = die.rolls
10.  c = Counter(rolls)
11.  c_sorted = sorted(c.items())
12.
13.  print(c_sorted)
```

Notice that we no longer have to keep a list of rolls as we did in the solution to the challenge in the **Adding a `roll()` Method** exercise (see page 165), which contains this code:

```
rolls = []
for i in range(100000):
    roll = die.roll()
rolls.append(roll)
```

Instead, we use the die's own `rolls` property:

```
rolls = die.rolls
```

5.8. Documenting Classes

One nice thing about using classes is the documentation you get for free. Check out the documentation for our `Circle` class in `classes-objects/Demos/Circle4.py`:

```
>>> import Circle4
>>> help(Circle4)
Help on module Circle4:

NAME
    Circle4

CLASSES
    builtins.object
        Circle

    class Circle(builtins.object)
     |  Circle(val, prop='r')
     |
     |  Methods defined here:
     |
     |  __init__(self, val, prop='r')
     |      Initialize self.  See help(type(self)) for accurate signature.
     |
     |  resize_by(self, amount)
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)
     |
     |  area
     |
     |  circumference
     |
     |  diameter
     |
     |  radius
```


This tells us that the `Circle` class has two methods: `__init__()` and `resize_by()` and four properties: `area`, `circumference`, `diameter`, and `radius`.

❖ 5.8.1. Using docstrings

Assuming we named our attributes, methods, and properties well, we will get some pretty good free documentation, but we can make it a lot better by using *docstrings*. A docstring is just a string placed at the beginning of a module, function, class, or method definition. The string can be a single line (in single quotes) or multiple lines (in triple quotes). By convention, double quotation marks (" or """) are used for docstrings.

As a rule, all classes and their methods should include docstrings. Methods should include documentation on any keyword arguments.

Following are some docstrings for our `Circle` class:

Demo 5.12: classes-objects/Demos/Circle5.py

```
1.   class Circle:
2.       "A circle"
3.       def __init__(self, val, prop='r'):
4.           """Create a circle based on a radius, diameter,
5.               circumference, or area
6.
7.           Keyword arguments:
8.           val (float) -- the value of prop
9.           prop (str)
10.              -- 'r' : radius (default)
11.              -- 'd' : diameter
12.              -- 'c' : circumference
13.              -- 'a' : area
14.          """
15.          self._radius = None
16.          self._diameter = None
17.          self._circumference = None
18.          self._area = None
19.          if prop == 'r':
20.              self.radius = val
21.          elif prop == 'd':
22.              self.diameter = val
23.          elif prop == 'c':
24.              self.circumference = val
25.          elif prop == 'a':
26.              self.area = val
27.          else:
28.              raise Exception('prop must be r, d, c, or a')
29.
30.      @property
31.      def radius(self):
32.          "radius of the circle object"
33.          return self._radius
34.
35.      @radius.setter
36.      def radius(self, r):
37.          """sets _radius, _diameter, _circumference, and _area of
38.     circle object"""
39.          self._radius = r
40.          self._diameter = r * 2
41.          self._circumference = r * 2 * math.pi
42.          self._area = r ** 2 * math.pi
43.
44.      @property
```

```
45.      def diameter(self):
46.          "diameter (2 x r) of the circle object"
47.          return self._diameter
48.
49.      @diameter.setter
50.      def diameter(self, d):
51.          """uses diameter d to set radius, which then
52.           updates all related pseudo-private attributes"""
53.          self.radius = d / 2
54.
55.      @property
56.      def circumference(self):
57.          "circumference (PI x d) of the circle object"
58.          return self._circumference
59.
60.      @circumference.setter
61.      def circumference(self, c):
62.          """uses circumference c to set radius, which then updates
63.           all related pseudo-private attributes"""
64.          self.radius = c / (2 * math.pi)
65.
66.      @property
67.      def area(self):
68.          "area (PI x r squared) of the circle object"
69.          return self._area
70.
71.      @area.setter
72.      def area(self, a):
73.          """uses area a to set radius, which then updates all
74.           related pseudo-private attributes"""
75.          self.radius = (a / math.pi) ** .5
76.
77.      def resize_by(self, amount):
78.          """resizes radius, which then updates all related
79.     pseudo-private attributes
80.
81.          Keyword arguments:
82.          amount (float) -- the amount by which to resize the radius
83.                            -- a negative number shrinks the radius
84.          """
85.          self.radius = self.radius * (1 + amount)
```

Check out how this improves the help documentation:

```
>>> import Circle5
>>> help(Circle5)
Help on module Circle5:

NAME
    Circle5

CLASSES
    builtins.object
        Circle

    class Circle(builtins.object)
     |  Circle(val, prop='r')
     |
     |  A circle
     |
     |  Methods defined here:
     |
     |  __init__(self, val, prop='r')
     |      Create a circle based on a radius, diameter,
     |          circumference, or area
     |
     |      Keyword arguments:
     |      val (float) -- the value of prop
     |      prop (str)
     |          -- 'r' : radius (default)
     |          -- 'd' : diameter
     |          -- 'c' : circumference
     |          -- 'a' : area
     |
     |  resize_by(self, amount)
     |      resizes radius, which then updates all related
     |                      pseudo-private attributes
     |
     |      Keyword arguments:
     |      amount (float) -- the amount by which to resize the radius
     |                     -- a negative number shrinks the radius
     |
```


Notice the help does not include the setter documentation. If you want that documentation to show up in the help, you could include it in the getter, like this:

```
@property
def radius(self):
    """radius of the circle object
    setter: sets _radius, _diameter, _circumference,
            and _area of Circle object
    """

@property
def diameter(self):
    """diameter (2 x r) of the circle object
    setter: uses diameter d to set radius, which then updates
            all pseudo-private attributes
    """
```

Exercise 17: Documenting the Die Class

10 to 20 minutes

In this exercise, you will add docstrings to the `Die` class.

1. Open `classes-objects/Exercises/Die.py` in your editor.
2. Add docstrings to the class so that `help(Die)` will output the following:

```
Help on class Die in module Die:

class Die(builtins.object)
|  A die
|
|  Methods defined here:
|
|  __init__(self, sides=6)
|      Creates a new standard die
|
|      Keyword arguments:
|          sides (int) -- number of die sides.
|
|  roll(self)
|      Returns a value between 1 and the number of die sides.
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
|
|  rolls
|      history of rolls
```

Solution: classes-objects/Solutions/Die2.py

```
1.    import random
2.
3.    class Die:
4.        "A die"
5.        def __init__(self, sides=6):
6.            """Creates a new standard die
7.
8.            Keyword arguments:
9.                sides (int) -- number of die sides."""
10.           if type(sides) != int or sides < 1:
11.               raise Exception('sides must be a positive integer.')
12.           self._sides = sides
13.           self._rolls = []
14.
15.       @property
16.       def rolls(self):
17.           "history of rolls"
18.           return self._rolls
19.
20.       def roll(self):
21.           "Returns a value between 1 and the number of die sides."
22.           roll = random.randint(1, self._sides)
23.           self._rolls.append(roll)
24.           return roll
```


5.9. Inheritance

Often you will find a class has a lot of the functionality you need, but is missing something. No worries. You can create your own class that inherits all the functionality of the other class and then you can make additions and modifications. The syntax for doing so is:

```
class A:
    pass

class B(A):
    pass
```

`B` is a *subclass* of `A`. `A` is a *superclass* of `B`.

❖ 5.9.1. Overriding a Class Method

Here is an example in which `B` overrides a method of `A`:

Demo 5.13: classes-objects/Demos/inheritance.py

```
1.   class A:
2.       def __init__(self, name):
3.           self.name = name
4.
5.       def intro(self):
6.           print('Hello, my name is {}.'.format(self.name))
7.
8.       def outro(self):
9.           print('Goodbye!')
10.
11.  class B(A):
12.      def intro(self):
13.          print('Hi, I am {}.'.format(self.name))
14.
15.  a = A('George')
16.  b = B('Ringo')
17.
18.  a.intro()
19.  b.intro()
20.  a.outro()
21.  b.outro()
```


The output will be:

```
…/classes-objects/Demos> python inheritance.py
Hello, my name is George.
Hi, I am Ringo.
Goodbye!
Goodbye!
```

Things to notice:

1. When we call `b.intro()`, it uses the `intro()` method defined in the `B` class.
2. When we call `b.outro()`, it uses the `outro()` method defined in the `A` class, because that method is not overwritten in the `B` class.

❖ 5.9.2. Extending a Class

The built-in `list` class has an `append()` method for appending new items to a `list`. It also has an `insert()` method for inserting new items at a specific index. However, it has no `prepend()` method. If you want to prepend an item to a list, the syntax is `mylist.insert(0, item)`. Let's create our own subclass of list that includes a `prepend()` method:

Demo 5.14: classes-objects/Demos/MyList.py

```
1.   class MyList(list):
2.       "A subclass of list with additional functionality"
3.       def prepend(self, obj):
4.           """prepend obj to list
5.
6.           Keyword arguments:
7.           obj -- obj to prepend"""
8.           self.insert(0, obj)
```


Now, let's test our new class:

```
>>> from MyList import MyList
>>> mylist = MyList(['a','b','c'])
>>> mylist.append('y')
>>> mylist.prepend('z')
>>> mylist
['z', 'a', 'b', 'c', 'y']
```

Notice that `mylist` has the `append()` method, which `MyList` inherited from `list` and it also has the new `prepend()` method.

Exercise 18: Extending the Die Class

15 to 25 minutes

In this exercise, you will create a `WeightedDie` class that extends the `Die` class.

Exercise Code 18.1: classes-objects/Exercises/WeightedDie.py

```
1.   import random
2.   from Die import Die
3.
4.   class WeightedDie(Die):
5.       "A weighted die"
6.       def __init__(self, weights, sides=6):
7.           """Creates a new weighted die
8.
9.           Keyword arguments:
10.          sides (int) -- number of die sides.
11.          weights (list) -- a list of integers holding the weights
12.              for each die side
13.          """
14.          if len(weights) != sides:
15.              raise Exception(f'weights must be a list of length {sides}.')
16.          super().__init__(sides)
17.          self._weights = weights
18.
19.      def roll(self):
20.          """Returns a value between 1 and the number of die sides."""
21.
22.          # COMPLETE THIS CODE
```


1. Open `classes-objects/Exercises/WeightedDie.py` in your editor.
2. Review the `__init__()` method of the `WeightedDie` class. Notice that it creates a pseudo-private `_weights` attribute that holds a list of weights, each corresponding to a side of the die. A six-sided die with the following `_weights` should roll a 6 five of every ten rolls (on average):

   ```
   [1, 1, 1, 1, 1, 5]
   ```

3. Complete the `roll()` method in the `WeightedDie` class. Again, the odds of returning a value should correlate to the weight in `self._weights`. One way of doing this is to create a new

list that contains each possible roll `n` times, where `n` is the associated weight. For example, for the `self._weights` above, the new list would look like this:

```
[1, 2, 3, 4, 5, 6, 6, 6, 6, 6]
```

Then use `random.choice()` to select a value from that list.

4. To test your solution, run the following code:

```
from WeightedDie import WeightedDie
from collections import Counter
die = WeightedDie(weights=[1, 1, 1, 1, 1, 5])

for i in range(100000):
    roll = die.roll()

c = Counter(die.rolls)
c_sorted = sorted(c.items())
c_sorted
```


The output should be something like this:

```
[(1, 9899), (2, 10012), (3, 10083), (4, 10133), (5, 10011), (6, 49862)]
```

5. Notice that the instance of `WeightedDie` has a `rolls` property, which it inherited from the `Die` class.

Solution: classes-objects/Solutions/WeightedDie.py

```
1.  import random
2.
3.  from Die2 import Die
4.
5.  class WeightedDie(Die):
6.      "A weighted die"
7.      def __init__(self, weights, sides=6):
8.          """Creates a new weighted die
9.
10.         Keyword arguments:
11.         sides (int) -- number of die sides.
12.         weights (list) -- a list of integers holding the weights
13.             for each die side
14.         """
15.         if len(weights) != sides:
16.             raise Exception(f'weights must be a list of length {sides}.')
17.         super().__init__(sides)
18.         self._weights = weights
19.
20.     def roll(self):
21.         """Returns a value between 1 and the number of die sides."""
22.         options = []
23.         for i in range(self._sides):
24.             for j in range(self._weights[i]):
25.                 options.append(i+1)
26.         roll = random.choice(options)
27.         self._rolls.append(roll)
28.         return roll
```


5.10. Extending a Class Method

Suppose a class you are extending has a method that does almost everything you want it to do, but you'd like to add something more. You may find you're able to extend the method rather than overwrite it. Here's a simple example:

Demo 5.15: classes-objects/Demos/extending_a_class_method.py

```
1.   class A:
2.       def __init__(self, name):
3.           self.name = name
4.
5.       def intro(self):
6.           print('Hello, my name is {}.'.format(self.name))
7.
8.       def outro(self):
9.           print('Goodbye!')
10.
11.  class B(A):
12.      def intro(self):
13.          super().intro()
14.          print('It\'s very nice to meet you.')
15.
16.  a = A('George')
17.  b = B('Ringo')
18.
19.  a.intro()
20.  print('-------')
21.  b.intro()
22.  print('-------')
23.  a.outro()
24.  b.outro()
```


This code produces the following output:

```
…/classes-objects/Demos> python extending_a_class_method.py
Hello, my name is George.
-------
Hello, my name is Ringo.
It's very nice to meet you.
-------
Goodbye!
Goodbye!
```

Notice the `intro()` method of class `B`. It first calls `super().intro()`, which calls the `intro()` method of the superclass. Then it extends the method by printing "It's very nice to meet you."

Creating a Non-Negative Counter

You may remember from a previous lesson (see page 23) that when subtracting from a `Counter`, it is possible to end up with negative counts:

Subtracting with a Counter

```
>>> c = Counter(['green', 'blue', 'blue', 'red', 'yellow', 'green', 'blue'])
>>> c # Before subtraction:
Counter({'blue': 3, 'green': 2, 'red': 1, 'yellow': 1})
>>> c.subtract(['red', 'yellow', 'yellow', 'purple'])
>>> c # After subtraction:
Counter({'blue': 3, 'green': 2, 'red': 0, 'yellow': -1, 'purple': -1})
```

Often, as with a product inventory, having a negative count doesn't make sense.

`Counter` objects have a special `__setitem__()` method that sets key values. We can override that method like this:

Demo 5.16: classes-objects/Demos/NonNegativeCounter.py

```
1.  from collections import Counter
2.
3.  class NonNegativeCounter(Counter):
4.      'Counter that disallows negative values'
5.      def __setitem__(self, key, value):
6.          value = 0 if value < 0 else value
7.          super().__setitem__(key, value)
```

On line 6, we set `value` to `0` if it is less than `0`. Otherwise, we set it to the `value` passed in to `__setitem__()`.

Then, on the next line, we pass `key` and `value` to `super().__setitem__()`.

`super()` refers to this class's *superclass*; that is, to `Counter`.

Let's see how it works:

```
>>> from NonNegativeCounter import NonNegativeCounter
>>> c = NonNegativeCounter(['green', 'blue', 'blue', 'red', 'yellow', 'green', 'blue'])
>>> c.subtract(['red', 'yellow', 'yellow', 'purple'])
>>> c
NonNegativeCounter({'blue': 3, 'green': 2, 'red': 0, 'yellow': 0, 'purple': 0})
```


Notice that this time the values for the `yellow` and `purple` keys are both *0*.

Exercise 19: Extending the roll() Method

10 to 20 minutes

In this exercise, you will create a `WeightingDie` class that extends the `WeightedDie` class. A weighting die starts with equal weights on each side, but it becomes weighted by giving more weight to rolls it has rolled before. It does this by modifying the `_weights` attribute with each roll. Here is the initial code:

Exercise Code 19.1: classes-objects/Exercises/WeightingDie.py

```
1.   import random
2.   from WeightedDie import WeightedDie
3.
4.   class WeightingDie(WeightedDie):
5.       "A weighting die"
6.       def __init__(self, sides=6):
7.           """Creates a die that favors sides it has previously rolled
8.
9.           Keyword arguments:
10.          sides (int) -- number of die sides.
11.          """
12.          self._weights = [1] * sides
13.          super().__init__(self._weights, sides)
14.
15.      def roll(self):
16.          """Returns a value between 1 and the number of die sides."""
17.         # COMPLETE THE CODE
```


Review the `__init__()` method of the `WeightingDie` class. Notice that it does not take a `weights` parameter like its superclass does. Rather, it sets all values in `_weights` to `1` using:

```
self._weights = [1] * sides
```

It then calls `super().__init__()` passing in `self._weights` and `sides`.

1. Open `classes-objects/Exercises/WeightingDie.py` in your editor.
2. Complete the `roll()` method so that it:
    A. Calls the `roll()` method of the superclass and stores the result in a local variable.

B. Modifies `self._weights` so that the weight for the roll just rolled is incremented by `1`.

C. Returns the roll.

3. To test your solution, run the following code:

```
from collections import Counter
from WeightingDie import WeightingDie
die = WeightingDie()

for i in range(10000):
    roll = die.roll()

c = Counter(die.rolls)
c_sorted = sorted(c.items())
c_sorted
```


The output should be something like this:

```
[(1, 473), (2, 70), (3, 828), (4, 4418), (5, 3359), (6, 852)]
```

Solution: classes-objects/Solutions/WeightingDie.py

```
1.    import random
2.    from WeightedDie import WeightedDie
3.
4.    class WeightingDie(WeightedDie):
5.        "A weighted die"
6.        def __init__(self, sides=6):
7.            """Creates a die that favors sides it has previously rolled
8.
9.            Keyword arguments:
10.           sides (int) -- number of die sides.
11.           """
12.           self._weights = [1] * sides
13.           super().__init__(self._weights, sides)
14.
15.       def roll(self):
16.           """Returns a value between 1 and the number of die sides."""
17.           result = super().roll()
18.           self._weights[result-1] += 1
19.           return result
```


5.11. Static Methods

As we've discussed, when you call a regular method on an instance of a class, the instance itself is passed in as the first argument. Python classes can also have *static methods*, which are created by including `@staticmethod` immediately before the method. A static method can be called directly on the class or on an instance of the class. It does not take `self` as the first parameter. For instance:

```
class Planet:
    @staticmethod
    def greet():
        print('hello')

Planet.greet()
earth = Planet()
earth.greet()
```

And here is a more practical example:

Demo 5.17: classes-objects/Demos/Triangle.py

```
1.   class Triangle:
2.       def __init__(self, sides):
3.           if not self.is_triangle(sides):
4.               raise Exception('Cannot make triangle with those sides.')
5.           self._sides = sides
6.
7.       @property
8.       def perimeter(self):
9.           return sum(self._sides)
10.
11.      @property
12.      def area(self):
13.          p = self.perimeter/2
14.          a = self._sides[0]
15.          b = self._sides[1]
16.          c = self._sides[2]
17.          return ( p * (p-a) * (p-b) * (p-c) ) ** .5
18.
19.      @staticmethod
20.      def is_triangle(sides):
21.          if len(sides) != 3:
22.              return False
23.          sides.sort()
24.          if sides[0] + sides[1] < sides[2]:
25.              return False
26.          return True
```


Things to notice:

1. The `is_triangle()` method is preceded by `@staticmethod`, indicating that it is a static method.
2. The `is_triangle()` does **not** take self as its first parameter.
3. On line 3, the `__init__()` method makes use of the `is_triangle()` method to check whether it is possible to make a triangle from the sides. It can call the method on `self` (i.e., the instance) but doesn't pass `self` in. It just passes in `sides`.

Let's try out the `Triangle` class:

```
>>> from Triangle import Triangle
>>> good = [3,3,5]
>>> bad = [3,3,9]
>>> print( Triangle.is_triangle(good) )
True
>>> print( Triangle.is_triangle(bad) )
False
>>> t1 = Triangle(good)
>>> print(t1.area)
4.14578098794425
>>> t2 = Triangle(bad)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Webucator\Python\classes-objects\Demos\Triangle.py", line 4, in __init__
    raise Exception('Cannot make triangle with those sides.')
Exception: Cannot make triangle with those sides.
```

Things to note:

1. The static `is_triangle()` method of the `Triangle` class returns `True` for a good triangle and `False` for a bad triangle.
2. We call the static method using the class name, `Triangle`. We do not need to instantiate a `triangle` object to call the method.
3. We can instantiate and print the area of a good triangle but if we attempt to instantiate a bad triangle, an exception is raised in the `__init__` method.

5.12. Class Attributes and Methods

❖ 5.12.1. Class Attributes

A class attribute is an attribute that is defined outside of a method, like `foo` is in the following code:

```
class A:
    foo = 1
    def __init__(self):
        pass
```

Class attributes can be accessed by classes themselves and by their instance objects; however, you have to be careful, as instance attributes will take precedence over class attributes. The following code illustrates this:

```
>>> class A:
...     foo = 1
...     bar = 1
...     def __init__(self):
...             self.foo = 2
...
>>> a = A()
>>> print(a.foo, A.foo, a.bar, A.bar)
2 1 1 1
```

Notice that `a.foo` and `A.foo` return different values, but `a.bar` and `A.bar` both return the value of the class attribute.

However, if you use an instance variable to modify a mutable object stored as a class attribute, a new object will not be created. To see this, examine the following code:


```
>>> class A:
...     foo = ['a','b','c']
...     def __init__(self):
...         self.foo.append('d')
...
>>> a = A()
>>> print(a.foo, A.foo)
['a', 'b', 'c', 'd'] ['a', 'b', 'c', 'd']
```

In this case, `a.foo` and `A.foo` return the same values, indicating that they are pointing to the same object.

❖ 5.12.2. Class Methods

Class methods are different from standard methods, which receive the instance object as their first parameter, and static methods, which do not receive any default arguments. Class methods receive the class itself as their first argument. To indicate that a method is a class method, precede it with `@classmethod`. Just like with a static method, a class method can be called directly on the class or on an instance of the class. Consider the following class:

Demo 5.18: classes-objects/Demos/MyCounter.py

```
1.   class MyCounter:
2.       count = 1
3.
4.       @classmethod
5.       def increment_count(cls):
6.           cls.count += 1
7.           return cls.count
```

Let's import this class at the Python terminal and make some calls to `increment_count()`:

```
>>> from MyCounter import MyCounter
>>> MyCounter.increment_count()
2
>>> c1 = MyCounter()
>>> c1.increment_count()
3
>>> c2 = MyCounter()
>>> c2.increment_count()
4
>>> MyCounter.increment_count()
5
```


Notice that the count is kept by the class itself and it increments every time `increment_count()` is called on the class or on any of its instances.

self and cls Parameters

The parameter names `self` and `cls` used in standard and class methods are used by convention. You can name them whatever you want, but it's best to stick with the convention.

So, when do you use class attributes and methods? Imagine you have a `Plane` class. Instances of `Plane` can take off and land. You want to keep track of all the `Plane` instances that have been created and also know how many are currently in the air. Here's a class for doing that:

Demo 5.19: classes-objects/Demos/Plane1.py

```
1.    class Plane:
2.        planes = []
3.        def __init__(self):
4.            self._in_air = False
5.            self.planes.append(self)
6.
7.        def take_off(self):
8.            self._in_air = True
9.
10.       def land(self):
11.           self._in_air = False
12.
13.       @classmethod
14.       def num_planes(cls):
15.           return len(cls.planes)
16.
17.       @classmethod
18.       def num_planes_in_air(cls):
19.           return len([plane for plane in cls.planes if plane._in_air])
```


Let's test the `Plane` class with the following statements:

```
>>> from Plane1 import Plane
>>> p1 = Plane()
>>> p2 = Plane()
>>> p3 = Plane()
>>> p1.take_off()
>>> p2.take_off()
>>> p1.land()
>>> print(Plane.num_planes(), Plane.num_planes_in_air())
3 1
```

Things to note:

1. When we append `self` to `self.planes` (line 5), the `Plane` instance actually gets appended to the class attribute `planes`. That's because there is no instance attribute `planes`, so the instance looks to the class for the attribute.
2. `num_planes()` and `num_planes_in_air()` are both class methods, which is why we can call them on `Plane` (e.g., `Plane.num_planes()` and `Plane.num_planes_in_air()`).
3. After two planes take off and one lands, there is only one plane left in the air.

❖ 5.12.3. You Must Consider Subclasses

There is a hard-to-spot problem in our code though. It only reveals itself when we subclass `Plane`, like this :

Demo 5.20: classes-objects/Demos/Jet.py

```
1.   from Plane1 import Plane
2.
3.   class Jet(Plane):
4.       def __init__(self):
5.           self.planes = []
6.           super().__init__()
```

Watch what happens when we run the following code:

```
>>> from Plane1 import Plane
>>> from Jet import Jet
>>> p1 = Plane()
>>> p2 = Plane()
>>> p3 = Plane()
>>> p1.take_off()
>>> p2.take_off()
>>> p1.land()
>>> p4 = Jet()
>>> p4.take_off()
>>> print(Plane.num_planes(), Plane.num_planes_in_air())
3 1
```


Notice that there are a total of 3 planes with 1 in the air, but we created a fourth plane, a jet, and had it take off. Why isn't it counted?

Notice that the `__init__()` method of `Jet` defines a `self.planes` attribute. When we append `self` to `self.planes` in the `__init__()` method of the superclass, this plane instance gets appended to the instance attribute `planes`. But our class methods are looking at the class attribute `planes`, so our `Jet` objects won't be included, which is why the results still show a total of three planes with one in the air.

The solution is to change the `__init__()` method of the superclass. One possibility is to use `Plane` instead of `self` when appending the object to `planes`:

```
def __init__(self):
   self._in_air = False
   Plane.planes.append(self)
```

But it's better not to use the class name within the class definition. Instead, you can do this:

```
def __init__(self):
   self._in_air = False
   type(self).planes.append(self)
```

`type(self)` returns the class of the instance object, which is exactly what we want. Now, when `Jet` objects are initialized, they will be appended to the class attribute `planes` rather than the instance attribute `planes`.

After making this change, run the same code again and you will see that the jet gets counted.

5.13. Abstract Classes and Methods

An abstract class is a class that cannot be instantiated but is created for the purposes of subclassing. For example, imagine we're creating a game with a bunch of flying objects, including planes and birds. A few things to note:

1. Planes can only land when they are over land.
2. Birds can only take off when their wings are healthy.
3. Birds can land anywhere.

Both birds and planes can take off and land, but they may do so in different ways and they may differ in other ways as well. So, we'll create a `FlyingObject` class for objects that can fly, but we don't want developers to be able to instantiate `FlyingObject` objects. Rather, we want them to subclass `FlyingObject` to create more specific object types.

Here's our first stab at our `FlyingObject` class:

Demo 5.21: classes-objects/Demos/FlyingObject1.py

```
1.    class FlyingObject():
2.        _flyingobjects = []
3.        def __init__(self, name):
4.            self._in_air = False
5.            self.name = name
6.            type(self)._flyingobjects.append(self)
7.
8.        def take_off(self):
9.            self._in_air = True
10.
11.       def land(self):
12.           self._in_air = False
13.
14.       @classmethod
15.       def flying_objects(cls):
16.           return cls._flyingobjects
17.
18.       def __str__(self):
19.           return self.name
```


Let's test our new class:

```
>>> from FlyingObject1 import FlyingObject
>>> ufo = FlyingObject('UFO')
>>> print(ufo.flying_objects()[0])
UFO
```

For the most part, this class serves our purposes, but it has one flaw: *it can be instantiated*, as the output from the preceding code reveals.

The __str__(self) Method

The `__str__(self)` method is a special method that is called when an object is implicitly converted to a string. Because the `FlyingObject` class's `__str__()` method returns the `name` attribute, `print(ufo.flying_objects()[0])` outputs "UFO".

To prevent our class from being instantiated, we need to explicitly specify that `FlyingObject` is an *abstract class*. The way to do that is to make any one of its methods abstract. Doing so, will:

1. Prevent developers from instantiating `FlyingObject` objects.
2. Force subclasses to implement the methods marked abstract.

In Python, you create an abstract class, by:

1. Importing the `abc` module.
2. Creating the class using `metaclass=abc.ABCMeta`.
3. Using `@abc.abstractmethod` decorators.

Here again is our `FlyingObject` class, now legitimately abstract:

Demo 5.22: classes-objects/Demos/FlyingObject.py

```
1.   import abc
2.
3.   class FlyingObject(metaclass=abc.ABCMeta):
4.       _flyingobjects = []
5.       def __init__(self, name):
6.           self._in_air = False
7.           self.name = name
8.           type(self)._flyingobjects.append(self)
9.
10.      @abc.abstractmethod
11.      def take_off(self):
12.          self._in_air = True
13.
14.      @abc.abstractmethod
15.      def land(self):
16.          self._in_air = False
17.
18.      @classmethod
19.      def flying_objects(cls):
20.          return cls._flyingobjects
21.
22.      def __str__(self):
23.          return self.name
```


Now, we get an error when we attempt to create an instance of `FlyingObject`:

```
>>> from FlyingObject import FlyingObject
>>> ufo = FlyingObject('UFO')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class FlyingObject with abstract methods land,
take_off
```

We also get an error if we attempt to create (and then instantiate) a subclass of `FlyingObject` without implementing all the abstract methods:

Demo 5.23: classes-objects/Demos/Plane2.py

```
1.    from FlyingObject import FlyingObject
2.
3.    class Plane(FlyingObject):
4.
5.        @property
6.        def pilot_awake(self):
7.            return True
8.        def take_off(self):
9.            if self.pilot_awake:
10.               super().take_off()
11.           self._in_air = True
```


Let's try to instantiate a `Plane` object:

```
>>> from Plane2 import Plane
>>> plane = Plane()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Plane with abstract methods land
```

As you can see, the attempt to create a `Plane` instance fails because the Plane class doesn't define a `land()` method.

We can fix the problem with the `Plane` class by defining the `land()` method:

Demo 5.24: classes-objects/Demos/Plane3.py

```
1.    from FlyingObject import FlyingObject
2.
3.    class Plane(FlyingObject):
4.        _planes = []
5.        _flying_objects = []
6.        def __init__(self, name):
7.            super().__init__(name)
8.            type(self)._planes.append(self)
9.            self._over_land = True
10.
11.       def take_off(self):
12.           super().take_off()
13.
14.       def land(self):
15.           if self.over_land:
16.               super().land()
17.
18.       @classmethod
19.       def planes(cls):
20.           return cls._planes
21.
22.       @property
23.       def over_land(self):
24.           return self._over_land
25.
26.       @over_land.setter
27.       def over_land(self, over_land):
28.           self._over_land = over_land
```


Now, we can successfully instantiate a `Plane` object:

```
>>> from Plane3 import Plane
>>> plane = Plane('Air Force One')
>>> print(FlyingObject.flying_objects()[0])
Air Force One
```

And, just to be complete, here's our `Bird` class:

Demo 5.25: classes-objects/Demos/Bird.py

```
1.    from FlyingObject import FlyingObject
2.
3.    class Bird(FlyingObject):
4.        _birds = []
5.        def __init__(self, name):
6.            super().__init__(name)
7.            type(self)._birds.append(self)
8.            self._healthy_wings = True
9.
10.       def take_off(self):
11.           if self.healthy_wings:
12.               super().take_off()
13.
14.       def land(self):
15.           super().land()
16.
17.       @classmethod
18.       def birds(cls):
19.           return cls._birds
20.
21.       @property
22.       def healthy_wings(self):
23.           return self._healthy_wings
24.
25.       @healthy_wings.setter
26.       def healthy_wings(self, healthy):
27.           self._healthy_wings = healthy
```


The following script makes use of both the `Bird` and `Plane` classes:

Demo 5.26: classes-objects/Demos/flying_objects.py

```
1.   from Plane import Plane3
2.   from Bird import Bird
3.   from FlyingObject import FlyingObject
4.
5.   p1 = Plane('Spirit of St. Louis')
6.   p2 = Plane('Air Force One')
7.   b1 = Bird('Big Bird')
8.   b2 = Bird('Roadrunner')
9.   b2.healthy_wings = False
10.  b3 = Bird('Tweety')
11.
12.  p1.take_off()
13.  p1.over_land = False
14.  p1.land()
15.
16.  b1.take_off()
17.  b2.take_off()
18.
19.  print('Flying Objects:')
20.  for object in p1.flying_objects():
21.      print(object)
22.  print('-----------')
23.
24.  print('Planes:')
25.  for plane in p1.planes():
26.      print(plane)
27.  print('-----------')
28.
29.  print('Birds:')
30.  for bird in b1.birds():
31.      print(bird)
```


Beginning on line 20, we iterate through `p1.flying_objects()`, but we could iterate through `flying_objects()` on any `Plane` or `Bird` object to see all the instances of `FlyingObject` subclasses.

Run this file in the terminal. You should get the following results:

```
…/classes-objects/Demos> python flying_objects.py
Flying Objects:
Spirit of St. Louis
Air Force One
Big Bird
Roadrunner
Tweety
-----------
Planes:
Spirit of St. Louis
Air Force One
-----------
Birds:
Big Bird
Roadrunner
Tweety
```

5.14. Understanding Decorators

We have mentioned several *decorators* in this lesson but have not explained what a decorator is. Decorators are functions that add functionality to (i.e., "decorate") other functions. They take a function as an argument and return a different function. Remember that functions are objects and objects can be passed from function to function. The following code illustrates this:

Demo 5.27: classes-objects/Demos/decorator1.py

```
1.   def foo(f):
2.       print(f)
3.       def foo_inner():
4.           pass
5.       return foo_inner
6.
7.   def bar():
8.       pass
9.
10.  print(bar)
11.  bar = foo(bar)
12.  print(bar)
```

Running this file will output:

```
…/classes-objects/Demos> python .\decorator1.py
<function bar at 0x0129B538>
<function bar at 0x0129B538>
<function foo.<locals>.foo_inner at 0x0129B4F0>
```

1. On line 10, we print `bar` and see that it is a function object. `0x0129B538` is the object's unique memory address.
2. On line 11, we call `foo()` and pass it the `bar` function object. The `foo()` function prints out the object (line 2). And we can see that it prints exactly the same thing, meaning that the local variable `f` is pointing to the `bar()` function defined on lines 7 and 8.
3. On lines 3 and 4, we define `foo_inner()`, which is a function local to the `foo()` function, meaning it can only be called from within `foo()`, unless `foo()` returns it, which it does.
4. Back on line 11, we overwrite the `bar` variable with whatever `foo()` returns, which is the the `foo_inner()` function.
5. On line 12, we print `bar` and see that it now contains a different function: the local `foo_inner()` function. Because `bar` is global, `foo_inner()` can now be called from anywhere.

The main takeaway from this is that functions can be passed around just like any other object.

Now, let's take a look at an example of how we can decorate a function with a decorator:

Demo 5.28: classes-objects/Demos/decorator2.py

```
1.   from datetime import datetime
2.
3.   def format_report(f):
4.       def inner(text):
5.           print('MY REPORT')
6.           print('-' * 50)
7.           f(text)
8.           print('-' * 50)
9.           print('Report completed: {}.'.format(datetime.now()))
10.      return inner
11.
12.  def report(text):
13.      print(text)
14.
15.  report = format_report(report)
16.
17.  report('I have created my first decorator.')
```

Here is the output:

```
…/classes-objects/Demos> python decorator2.py
MY REPORT
--------------------------------------------------
I have created my first decorator.
--------------------------------------------------
Report completed: 2020-04-01 20:16:55.607922.
```

1. Look at the `report()` function on lines 12 and 13. All it does is print the text that is passed in.
2. On line 15, the `report` function is passed to `format_report()`, which defines and later returns an `inner()` function. This `inner()` function:
    A. Prints a couple of lines of text.
    B. Runs the passed-in function.
    C. Prints another couple of lines of text.
3. Back on line 15, we overwrite the `report` variable with the function object returned by `format_report()`.
4. Now, on line 17, when we call `report()`, it runs the `inner()` function returned by `format_report()`.

Can you see how `format_report()` is decorating (i.e., adding functionality to) the `report()` function? Of course, `format_report()` doesn't do anything terribly exciting, but a decorator can do anything you want it to do. For example, it could keep an event log or send an email.

There is a special decorator syntax, which you have already seen when creating properties and static, class and abstract methods. Instead of explicitly overwriting a function variable with the function returned by a decorator (e.g., `report = format_report(report)`), you indicate that the function will be decorated like this:

Demo 5.29: classes-objects/Demos/decorator3.py

```
1.   from datetime import datetime
2.
3.   def format_report(f):
4.       def inner(text):
5.           print('MY REPORT')
6.           print('-' * 50)
7.           f(text)
8.           print('-' * 50)
9.           print('Report completed: {}.'.format(datetime.now()))
10.      return inner
11.
12.  @format_report
13.  def report(text):
14.      print(text)
15.
16.  report('I have created my second decorator.')
```


Run this and you will see that it returns something similar to what `decorator2.py` returned:

```
…/classes-objects/Demos> python decorator3.py
MY REPORT
--------------------------------------------------
I have created my second decorator.
--------------------------------------------------
Report completed: 2020-04-01 21:18:35.833127.
```

This should give you a better understanding of how the `@property`, `@staticmethod`, `@classmethod`, and `@abc.abstractmethod` are working behind the scenes.

Conclusion

In this lesson, you have learned how to create Python classes and write object-oriented code.