2 The Standard Library

Lab Objective: *Python is designed to make it easy to implement complex tasks with little code. To that end, every Python distribution includes several built-in functions for accomplishing common tasks. In addition, Python is designed to import and reuse code written by others. A Python file with code that can be imported is called a* module*. All Python distributions include a collection of modules for accomplishing a variety of tasks, collectively called the* Python Standard Library*. In this lab we explore some built-in functions, learn how to create, import, and use modules, and become familiar with the standard library.*

Built-in Functions

Python has several built-in functions that may be used at any time. IPython's object introspection feature makes it easy to learn about these functions: start IPython from the command line and use `?` to bring up technical details on each function.

```
In [1]: min?
Docstring:
min(iterable, *[, default=obj, key=func]) -> value
min(arg1, arg2, *args, *[, key=func]) -> value

With a single iterable argument, return its smallest item. The
default keyword-only argument specifies an object to return if
the provided iterable is empty.
With two or more arguments, return the smallest argument.
Type:      builtin_function_or_method

In [2]: len?
Signature: len(obj, /)
Docstring: Return the number of items in a container.
Type:      builtin_function_or_method
```

Function	Returns
abs()	The absolute value of a real number, or the magnitude of a complex number.
min()	The smallest element of a single iterable, or the smallest of several arguments. Strings are compared based on lexicographical order: numerical characters first, then upper-case letters, then lower-case letters.
max()	The largest element of a single iterable, or the largest of several arguments.
len()	The number of items of a sequence or collection.
round()	A float rounded to a given precision in decimal digits.
sum()	The sum of a sequence of numbers.

Table 2.1: Common built-in functions for numerical calculations.

```
# abs() can be used with real or complex numbers.
>>> print(abs(-7), abs(3 + 4j))
7 5.0

# min() and max() can be used on a list, string, or several arguments.
# String characters are ordered lexicographically.
>>> print(min([4, 2, 6]), min("aXbYcZ"), min('1', 'a', 'A'))
2 X 1
>>> print(max([4, 2, 6]), max("aXbYcZ"), max('1', 'a', 'A'))
6 c a

# len() can be used on a string, list, set, dict, tuple, or other iterable.
>>> print(len([2, 7, 1]), len("abcdef"), len({1, 'a', 'a'}))
3 6 2

# sum() can be used on iterables containing numbers, but not strings.
>>> my_list = [1, 2, 3]
>>> my_tuple = (4, 5, 6)
>>> my_set = {7, 8, 9}
>>> sum(my_list) + sum(my_tuple) + sum(my_set)
45
>>> sum([min(my_list), max(my_tuple), len(my_set)])
10

# round() is particularly useful for formatting data to be printed.
>>> round(3.14159265358979323, 2)
3.14
```

See https://docs.python.org/3/library/functions.html for more detailed documentation on all of Python's built-in functions.

Problem 1. Write a function that accepts a list L and returns the minimum, maximum, and average of the entries of L in that order as multiple values (separated by a comma). Can you implement this function in a single line?

Namespaces

Whenever a Python object—a number, data structure, function, or other entity—is created, it is stored somewhere in computer memory. A *name* (or variable) is a reference to a Python object, and a *namespace* is a dictionary that maps names to Python objects.

```
# The number 4 is the object, 'number_of_students' is the name.
>>> number_of_sudents = 4

# The list is the object, and 'beatles' is the name.
>>> beatles = ["John", "Paul", "George", "Ringo"]

# Python statements defining a function also form an object.
# The name for this function is 'add_numbers'.
>>> def add_numbers(a, b):
...     return a + b
...
```

A single equals sign assigns a name to an object. If a name is assigned to another name, that new name refers to the same object as the original name.

```
>>> beatles = ["John", "Paul", "George", "Ringo"]
>>> band_members = beatles            # Assign a new name to the list.
>>> print(band_members)
['John', 'Paul', 'George', 'Ringo']
```

To see all of the names in the current namespace, use the built-in function `dir()`. To delete a name from the namespace, use the `del` keyword (**with caution!**).

```
# Add 'stem' to the namespace.
>>> stem = ["Science", "Technology", "Engineering", "Mathematics"]
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__', 'stem']

# Remove 'stem' from the namespace.
>>> del stem
>>> "stem" in dir()
False
>>> print(stem)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'stem' is not defined
```

Note

Many programming languages distinguish between *variables* and *pointers*. A pointer refers to a variable by storing the address in memory where the corresponding object is stored. Python names are essentially pointers, and traditional pointer operations and cleanup are done automatically. For example, Python automatically deletes objects in memory that have no names assigned to them (no pointers referring to them). This feature is called *garbage collection*.

Mutability

Every Python object type falls into one of two categories: a *mutable* object, which may be altered at any time, or an *immutable* object, which cannot be altered once created. Attempting to change an immutable object creates a new object in memory. If two names refer to the same mutable object, any changes to the object are reflected in both names since they still both refer to that same object. On the other hand, if two names refer to the same immutable object and one of the values is "changed," then one name will refer to the original object, and the other will refer to a new object in memory.

Achtung!

Failing to correctly copy mutable objects can cause subtle problems. For example, consider a dictionary that maps items to their base prices. To make a similar dictionary that accounts for a small sales tax, we might try to make a copy by assigning a new name to the first dictionary.

```
>>> holy = {"moly": 1.99, "hand_grenade": 3, "grail": 1975.41}
>>> tax_prices = holy                # Try to make a copy for processing.
>>> for item, price in tax_prices.items():
...     # Add a 7 percent tax, rounded to the nearest cent.
...     tax_prices[item] = round(1.07 * price, 2)
...
# Now the base prices have been updated to the total price.
>>> print(tax_prices)
{'moly': 2.13, 'hand_grenade': 3.21, 'grail': 2113.69}

# However, dictionaries are mutable, so 'holy' and 'tax_prices' actually
# refer to the same object. The original base prices have been lost.
>>> print(holy)
{'moly': 2.13, 'hand_grenade': 3.21, 'grail': 2113.69}
```

To avoid this problem, explicitly create a copy of the object by casting it as a new structure. Changes made to the copy will not change the original object, since they are distinct objects in memory. To fix the above code, replace the second line with the following:

```
>>> tax_prices = dict(holy)
```

Then, after running the same procedure, the two dictionaries will be different.

Problem 2. Determine which Python object types are mutable and which are immutable by repeating the following experiment for an int, str, list, tuple, and set.

1. Create an object of the given type and assign a name to it.
2. Assign a new name to the first name.
3. Alter the object via only one of the names (for tuples, use `my_tuple += (1,)`).
4. Check to see if the two names are equal. If they are, then changing one name also changes the other. Thus, both names refer to the same object and the object type is mutable. Otherwise, the names refer to different objects—meaning a new object was created in step 2—and therefore the object type is immutable.

For example, the following experiment shows that dict is a mutable type.

```
>>> dict_1 = {1: 'x', 2: 'b'}           # Create a dictionary.
>>> dict_2 = dict_1                     # Assign it a new name.
>>> dict_2[1] = 'a'                     # Change the 'new' dictionary.
>>> dict_1 == dict_2                    # Compare the two names.
True                                    # Both names changed!
```

Print a statement of your conclusions that clearly indicates which object types are mutable and which are immutable.

Achtung!

Mutable objects cannot be put into Python sets or used as keys in Python dictionaries. However, the values of a dictionary may be mutable or immutable.

```
>>> a_dict = {"key": "value"}             # Dictionaries are mutable.
>>> broken = {1, 2, 3, a_dict, a_dict}    # Try putting a dict in a set.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'

>>> okay = {1: 2, "3": a_dict}            # Try using a dict as a value.
```

Modules

A *module* is a Python file containing code that is meant to be used in some other setting, and not necessarily run directly.[1] The import statement loads code from a specified Python file. Importing a module containing some functions, classes, or other objects makes those functions, classes, or objects available for use by adding their names to the current namespace.

All import statements should occur at the top of the file, below the header but before any other code. There are several ways to use import:

1. `import <module>` makes the specified module available under the alias of its own name.

```
>>> import math                 # The name 'math' now gives
>>> math.sqrt(2)                # access to the math module.
1.4142135623730951
```

2. `import <module> as <name>` creates an alias for an imported module. The alias is added to the current namespace, but the module name itself is not.

```
>>> import numpy as np          # The name 'np' gives access to the numpy
>>> np.sqrt(2)                  # module, but the name 'numpy' does not.
1.4142135623730951
>>> numpy.sqrt(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'numpy' is not defined
```

3. `from <module> import <object>` loads the specified object into the namespace without loading anything else in the module or the module name itself. This is used most often to access specific functions from a module. The as statement can also be tacked on to create an alias.

```
>>> from random import randint  # The name 'randint' gives access to the
>>> r = randint(0, 10000)       # randint() function, but the rest of
>>> random.seed(r)              # the random module is unavailable.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'random' is not defined
```

In each case, the final word of the import statement is the name that is added to the namespace.

Running and Importing

Consider the following simple Python module, saved as `example1.py`.

```
# example1.py

data = list(range(4))
```

[1]Python files that are primarily meant to be executed, not imported, are often called *scripts*.

```python
def display():
    print("Data:", data)

if __name__ == "__main__":
    display()
    print("This file was executed from the command line or an interpreter.")
else:
    print("This file was imported.")
```

Executing the file from the command line executes the file line by line, including the code under the `if __name__ == "__main__"` clause.

```
$ python example1.py
Data: [0, 1, 2, 3]
This file was executed from the command line or an interpreter.
```

Executing the file with IPython's special `%run` command executes each line of the file and also adds the module's names to the current namespace. **This is the quickest way to test individual functions via IPython.**

```
In [1]: %run example1.py
Data: [0, 1, 2, 3]
This file was executed from the command line or an interpreter.

In [2]: display()
Data: [0, 1, 2, 3]
```

Importing the file also executes each line,[2] but only adds the indicated alias to the namespace. Also, code under the `if __name__ == "__main__"` clause is **not** executed when a file is imported.

```
In [1]: import example1 as ex
This file was imported.

# The module's names are not directly available...
In [2]: display()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-795648993119> in <module>()
----> 1 display()

NameError: name 'display' is not defined

# ...unless accessed via the module's alias.
In [3]: ex.display()
Data: [0, 1, 2, 3]
```

[2]Try importing the `this` or `antigravity` modules. Importing these modules actually executes some code.

Problem 3. Create a module called `calculator.py`. Write a function `sum()` that returns the sum of two arguments and a function `product()` that returns the product of two arguments. Also use `import` to add the `sqrt()` function from the `math` module to the namespace. When this file is either run or imported, nothing should be executed.

In your solutions file, import your new custom module. Write a function that accepts two numbers representing the lengths of the sides of a right triangle. Using only the functions from `calculator.py`, calculate and return the length of the hypotenuse of the triangle.

Achtung!

If a module has been imported in IPython and the source code then changes, using `import` again does **not** refresh the name in the IPython namespace. Use **run** instead to correctly refresh the namespace. Consider this example where we test the function `sum_of_squares()`, saved in the file `example2.py`.

```python
# example2.py

def sum_of_squares(x):
    """Return the sum of the squares of all positive integers
    less than or equal to x.
    """
    return sum([i**2 for i in range(1,x)])
```

In IPython, run the file and test `sum_of_squares()`.

```
# Run the file, adding the function sum_of_squares() to the namespace.
In [1]: %run example2

In [2]: sum_of_squares(3)
Out[2]: 5                           # Should be 14!
```

Since $1^2+2^2+3^2 = 14$, not 5, something has gone wrong. Modify the source file to correct the mistake, then run the file again in IPython.

```python
# example2.py

def sum_of_squares(x):
    """Return the sum of the squares of all positive integers
    less than or equal to x.
    """
    return sum([i**2 for i in range(1,x+1)])    # Include the final term.
```

```
# Run the file again to refresh the namespace.
In [3]: %run example2
```

```
# Now sum_of_squares() is updated to the new, corrected version.
In [4]: sum_of_squares(3)
Out[4]: 14                          # It works!
```

Remember that running or importing a file executes any freestanding code snippets, but any code under an `if __name__ == "__main__"` clause will **only** be executed when the file is run (not when it is imported).

The Python Standard Library

All Python distributions include a collection of modules for accomplishing a variety of common tasks, collectively called the *Python standard library*. Some commonly standard library modules are listed below, and the complete list is at `https://docs.python.org/3/library/`.

Module	Description
`cmath`	Mathematical functions for complex numbers.
`itertools`	Tools for iterating through sequences in useful ways.
`math`	Standard mathematical functions and constants.
`random`	Random variable generators.
`string`	Common string literals.
`sys`	Tools for interacting with the interpreter.
`time`	Time value generation and manipulation.

Use IPython's object introspection to quickly learn about how to use the various modules and functions in the standard library. Use `?` or `help()` for information on the module or one of its names. To see the entire module's namespace, use the `tab` key.

```
In [1]: import math

In [2]: math?
Type:        module
String form: <module 'math' (built-in)>
Docstring:
This module provides access to the mathematical functions
defined by the C standard.

# Type the module name, a period, then press tab to see the module's namespace.
In [3]: math.   # Press 'tab'.
    acos()      cos()       factorial() isclose()   log2()      tan()
    acosh()     cosh()      floor()     isfinite()  modf()      tanh()
    asin()      degrees()   fmod()      isinf()     nan         tau
    asinh()     e           frexp()     isnan()     pi          trunc()
    atan()      erf()       fsum()      ldexp()     pow()
    atan2()     erfc()      gamma()     lgamma()    radians()
    atanh()     exp()       gcd()       log()       sin()
    ceil()      expm1()     hypot()     log10()     sinh()
    copysign()  fabs()      inf         log1p()     sqrt()

In [3]: math.sqrt?
Signature: math.sqrt(x, /)
Docstring: Return the square root of x.
Type:      builtin_function_or_method
```

The Itertools Module

The `itertools` module makes it easy to iterate over one or more collections in specialized ways.

Function	Description
`chain()`	Iterate over several iterables in sequence.
`cycle()`	Iterate over an iterable repeatedly.
`combinations()`	Return successive combinations of elements in an iterable.
`permutations()`	Return successive permutations of elements in an iterable.
`product()`	Iterate over the Cartesian product of several iterables.

```
>>> from itertools import chain, cycle              # Import multiple names.

>>> list(chain("abc", ['d', 'e'], ('f', 'g')))      # Join several
['a', 'b', 'c', 'd', 'e', 'f', 'g']                 # sequences together.

>>> for i,number in enumerate(cycle(range(4))):     # Iterate over a single
...     if i > 10:                                  # sequence over and over.
...         break
...     print(number, end=' ')
...
0 1 2 3 0 1 2 3 0 1 2
```

A *k-combination* is a set of k elements from a collection where the ordering is unimportant. Thus the combination (a, b) and (b, a) are equivalent because they contain the same elements. One the other hand, a *k-permutation* is a sequence of k elements from a collection where the ordering matters. Even though (a, b) and (b, a) contain the same elements, they are counted as different permutations.

```
>>> from itertools import combinations, permutations

# Get all combinations of length 2 from the iterable "ABC".
>>> list(combinations("ABC", 2))
[('A', 'B'), ('A', 'C'), ('B', 'C')]

# Get all permutations of length 2 from "ABC". Note that order matters here.
>>> list(permutations("ABC", 2))
[('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'A'), ('C', 'B')]
```

Problem 4. The *power set* of a set A, denoted $\mathcal{P}(A)$ or 2^A, is the set of all subsets of A, including the empty set $\emptyset$ and A itself. For example, the power set of the set $A = \{a, b, c\}$ is $2^A = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$.

Write a function that accepts an iterable A. Use an `itertools` function to compute and return the power set of A as a list of sets (why couldn't it be a set of sets in Python?). The empty set should be returned as `set()`.

The Random Module

Many real-life events can be simulated by taking random samples from a probability distribution. For example, a coin flip can be simulated by randomly choosing between the integers 1 (for heads) and 0 (for tails). The `random` module includes functions for sampling from probability distributions and generating random data.

Function	Description
`choice()`	Choose a random element from a non-empty sequence, such as a list.
`randint()`	Choose a random integer integer over a closed interval.
`random()`	Pick a float from the interval [0, 1).
`sample()`	Choose several unique random elements from a non-empty sequence.
`seed()`	Seed the random number generator.
`shuffle()`	Randomize the ordering of the elements in a list.

Some of the most common `random` utilities involve picking random elements from iterables.

```
>>> import random

>>> numbers = list(range(1,11))      # Get the integers from 1 to 10.
>>> print(numbers)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> random.shuffle(numbers)          # Mix up the ordering of the list.
>>> print(numbers)                   # Note that shuffle() returns nothing.
[5, 9, 1, 3, 8, 4, 10, 6, 2, 7]

>>> random.choice(numbers)           # Pick a single element from the list.
5

>>> random.sample(numbers, 4)        # Pick 4 unique elements from the list.
[5, 8, 3, 2]

>>> random.randint(1,10)             # Pick a random number between 1 and 10.
10
```

The Time Module

The `time` module in the standard library include functions for dealing with time. In particular, the `time()` function measures the number of seconds from a fixed starting point, called "the Epoch" (January 1, 1970 for Unix machines).

```
>>> import time
>>> time.time()
1495243696.645818
```

The `time()` function is useful for measuring how long it takes for code to run: record the time just before and just after the code in question, then subtract the first measurement from the second to get the number of seconds that have passed.

```
>>> def time_for_loop(iters):
...     """Time how long it takes to iterate 'iters' times."""
...     start = time.time()         # Clock the starting time.
...     for _ in range(int(iters)):
...         pass
...     end = time.time()           # Clock the ending time.
...     return end - start          # Report the difference.
...
>>> time_for_loop(1e5)              # 1e5 = 100000.
0.005570173263549805
>>> time_for_loop(1e7)              # 1e7 = 10000000.
0.26819777488708496
```

The Sys Module

The sys (system) module includes methods for interacting with the Python interpreter. For our purposes, we care about the instance that you call a .py file in the command line or in your ipython terminal using either ‘python’ followed by the name of a .py file or ‘%run’ followed by the name of a .py file. One command we will use is sys.argv; which returns a list containing the .py file name and all arguments that were passed to the interpreter. This way we can interact with these initial arguments and execute certain parts of our code if the arguments satisfy certain conditions. This will be implemented in problem 5. Note, command line arguments are passed in after the name of the .py file and are separated by spaces like so:

```
# the 'python' command followed by a .py file followed by any arguments
$ python yourProgram.py argument1 argument2 argument3

# the equivalent form in an ipython terminal
In[1]: %run yourProgram.py argument1 argument2 argument3
```

Now we can execute code based on the command line arguments by inserting the use of the sys module in our .py file:

```
# example3.py
"""If there are two command line arguments  after the .py file  name, print a ↩
    descriptive statment."""
import sys

if len(sys.argv) == 3:  # if there's two arguments and the file name
    print("the first command line argument is " + sys.argv[1])
    print("the second command line argument is " + sys.argv[2])
else:
    print("um... I need two after the .py file. This is not two:")
    print(sys.argv)
```

Now provide command line arguments for the program to process.

```
# No extra command line arguments.
$ python example3.py
Output:
um... I need two after the .py file. This is not two:
['example3.py']

# With two arguments the if statement executes.
$ python example3.py crunchy juicy
Output:
the first command line argument is crunchy
the second command line argument is juicy
```

Note that the first command line argument is always the filename, so that is always the first element of the sys.argv list. This is why we have the if statement execute if the length of the list is 3 even though we want it to execute when there are 2 command line arguments. Also, `sys.argv` is always a list of strings. If a number is provided on the command line, it is converted to a string when it is stored in`sys.argv`. In IPython, command line arguments are specified after the `%run` command.

```
In [1]: %run example3.py 42 too many
Output:
um... I need two after the .py file. This is not two:
['example3.py', '42', 'too', 'many']
```

Another way to get input from the program user is to prompt the user for text. The built-in function `input()` pauses the program and waits for the user to type something. Like command line arguments, the user's input is parsed as a string.

```
>>> x = input("Enter a value for x: ")
Enter a value for x: 20             # Type '20' and press 'enter.'

>>> x
'20'                                # Note that x contains a string.

>> y = int(input("Enter an integer for y: "))
Enter an integer for y: 16          # Type '16' and press 'enter.'

>>> y
16                                  # Note that y contains an integer.
```

Problem 5. *Shut the box* is a popular British pub game that is used to help children learn arithmetic. The player starts with the numbers 1 through 9, and the goal of the game is to eliminate as many of these numbers as possible. At each turn the player rolls two dice, then chooses a set of integers from the remaining numbers that sum up to the sum of the dice roll. These numbers are removed, and the dice are then rolled again. The game ends when none of the remaining integers can be combined to the sum of the dice roll, and the player's final score is the sum of the numbers that could not be eliminated. For a demonstration, see `https://www.youtube.com/watch?v=mwURQC7mjDI`.

Modify your solutions file so that when the file is run with the correct command line arguments (but **not** when it is imported), the user plays a game of shut the box. The provided module `box.py` contains two functions that will be useful in your implementation of the game. You do not need to understand exactly how the functions work, but you do need to be able to import and use them correctly. Their functionality is outlined at the beginning of each function declaration. Your game should match the following specifications:

- Require three total command line arguments: the file name (included by default), the player's name, and a time limit in seconds. If there are not exactly three command line arguments, do not start the game.
- Track the player's remaining numbers, starting with 1 through 9.
- Use the `random` module to simulate rolling two six-sided dice. However, if the sum of the player's remaining numbers is 6 or less, roll only one die.
- The player wins if they have no numbers left, and they lose if they are out of time or if they cannot choose numbers to match the dice roll.
- If the game is not over, print the player's remaining numbers, the sum of the dice roll, and the number of seconds remaining. Prompt the user for numbers to eliminate. The input should be one or more of the remaining integers, separated by spaces. If the user's input is invalid, prompt them for input again before rolling the dice again.
  (Hint: use `round()` to format the number of seconds remaining nicely.)
- When the game is over, display the player's name, their score, and the total number of seconds since the beginning of the game. Congratulate or mock the player appropriately.

(Hint: **Before you start coding**, write an outline for the entire program, adding one feature at a time. Only start implementing the game after you are completely finished designing it.)

Your game should look similar to the following examples. The characters in red are typed inputs from the user.

```
$ python standard_library.py LuckyDuke 60

Numbers left: [1, 2, 3, 4, 5, 6, 7, 8, 9]
Roll: 12
Seconds left: 60.0
Numbers to eliminate: 3 9

Numbers left: [1, 2, 4, 5, 6, 7, 8]
```

```
Roll: 9
Seconds left: 53.51
Numbers to eliminate: 8 1

Numbers left: [2, 4, 5, 6, 7]
Roll: 7
Seconds left: 51.39
Numbers to eliminate: 7

Numbers left: [2, 4, 5, 6]
Roll: 2
Seconds left: 48.24
Numbers to eliminate: 2

Numbers left: [4, 5, 6]
Roll: 11
Seconds left: 45.16
Numbers to eliminate: 5 6

Numbers left: [4]
Roll: 4
Seconds left: 42.76
Numbers to eliminate: 4

Score for player LuckyDuke: 0 points
Time played: 15.82 seconds
Congratulations!! You shut the box!
```

The next two examples show different ways that a player could lose (which they usually do), as well as examples of invalid user input. Use the `box` module's `parse_input()` to detect invalid input.

```
$ python standard_library.py ShakySteve 10

Numbers left: [1, 2, 3, 4, 5, 6, 7, 8, 9]
Roll: 7
Seconds left: 10.0
Numbers to eliminate: Seven              # Must enter a number.
Invalid input

Seconds left: 7.64
Numbers to eliminate: 1, 2, 4            # Do not use commas.
Invalid input

Seconds left: 4.55
Numbers to eliminate: 1 2 3              # Numbers don't sum to the roll.
Invalid input
```

```
Seconds left: 2.4
Numbers to eliminate: 1 2 4

Numbers left: [3, 5, 6, 7, 8, 9]
Roll: 8
Seconds left: 0.31
Numbers to eliminate: 8
Game over!                              # Time is up!

Score for player ShakySteve: 30 points
Time played: 11.77 seconds
Better luck next time >:)
```

```
$ python standard_library.py SnakeEyesTom 10000

Numbers left: [1, 2, 3, 4, 5, 6, 7, 8, 9]
Roll: 2
Seconds left: 10000.0
Numbers to eliminate: 2

Numbers left: [1, 3, 4, 5, 6, 7, 8, 9]
Roll: 2
Game over!                              # Numbers cannot match roll.

Score for player SnakeEyesTom: 43 points
Time played: 1.53 seconds
Better luck next time >:)
```

Additional Material

More Built-in Functions

The following built-in functions are worth knowing, especially for working with iterables and writing very readable conditional statements.

Function	Description
`all()`	Return `True` if `bool(entry)` evaluates to `True` for *every* entry in the input iterable.
`any()`	Return `True` if `bool(entry)` evaluates to `True` for *any* entry in the input iterable.
`bool()`	Evaluate a single input object as `True` or `False`.
`eval()`	Execute a string as Python code and return the output.
`map()`	Apply a function to every item of the input iterable and return an iterable of the results.

```python
>>> from random import randint
# Get 5 random numbers between 1 and 10, inclusive.
>>> numbers = [randint(1,10) for _ in range(5)]

# If all of the numbers are less than 8, print the list.
>>> if all([num < 8 for num in numbers]):
...     print(numbers)
...
[1, 5, 6, 3, 3]

# If none of the numbers are divisible by 3, print the list.
>>> if not any([num % 3 == 0 for num in numbers]):
...     print(numbers)
...
```

Two-Player Shut the Box

Consider modifying your shut the box program so that it pits two players against each other (one player tries to shut the box while the other tries to keep it open). The first player plays a regular round as described in Problem 5. Suppose he or she eliminates every number but 2, 3, and 6. The second player then begins a round with the numbers 1, 4, 5, 7, 8, and 9, the numbers that the first player had eliminated. If the second player loses, the first player gets another round to try to shut the box with the numbers that the second player had eliminated. Play continues until one of the players eliminates their entire list. In addition, each player should have their own time limit that only ticks down during their turn. If time runs out on your turn, you lose no matter what.

Python Packages

Large programming projects often have code spread throughout several folders and files. In order to get related files in different folders to communicate properly, the associated directories must be organized into a Python *packages*. This is a common procedure when creating smart phone applications and other programs that have graphical user interfaces (GUIs).

A package is simply a folder that contains a file called `__init__.py`. This file is always executed first whenever the package is used. A package must also have a file called `__main__.py` in order to be executable. Executing the package will run `__init__.py` and then `__main__.py`, but importing the package will only run `__init__.py`.

Use the regular syntax to import a module or subpackage that is in the current package, and use `from <subpackage.module> import <object>` to load a module within a subpackage. Once a name has been loaded into a package's `__init__.py`, other files in the same package can load the same name with `from . import <object>`. To access code in the directory one level above the current directory, use the syntax `from .. import <object>` This tells the interpreter to go up one level and import the object from there. This is called an *explicit relative import* and cannot be done in files that are executed directly (like `__main__.py`).

Finally, to execute a package, run Python from the shell with the flag `-m` (for "module-name") and exclude the extension `.py`.

```
$ python -m package_name
```

See `https://docs.python.org/3/tutorial/modules.html#packages` for examples and more details.