Lab 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

Every Python installation comes with 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[, key=func]) -> value
min(a, b, c, ...[, key=func]) -> value

With a single iterable argument, return its smallest item.
With two or more arguments, return the smallest argument.
Type:      builtin_function_or_method

In [2]: len?
Docstring:
len(object) -> integer

Return the number of items of a sequence or collection.
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: Some 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 useful for formatting data to be printed.
>>> round(3.14159265358979323, 2)
3.14
```

More detailed documentation on all of Python's built-in functions can be found at `https://docs.python.org/2/library/functions.html`.

Problem 1. Write a function that accepts a list of numbers as input and returns a new list with the minimum, maximum, and average of the original list (in that order). Remember to use floating point division to calculate the average. Can you implement this function in a single line?

Namespaces

Names

All Python objects reside somewhere in computer memory. These objects may be numbers, data structures, functions, or any other sort of Python object. A *name* (or variable) is a reference to a Python object. 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 'students' is the name.
>>> students = ["John", "Paul", "George", "Ringo"]

# Python statements defining a function 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 that the original name refers to (or a copy of it—see the next section).

```
>>> students = ["John", "Paul", "George", "Ringo"]
>>> band_members = students
>>> 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 variable from the namespace, use the `del` keyword (be careful!).

```
>>> subjects = ["Statistics", "Technology", "Engineering", "Mathematics"]
>>> num = 4
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'num', 'subjects']
>>> del num
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'subjects']
>>> print(num)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'num' is not defined
```

Note

Many programming languages distinguish between *variables* and *pointers*. A pointer holds a memory location where an object is stored. Python names are essentially pointers. Objects in memory that have no names pointing to them are automatically deleted (this is called *garbage collection*).

Mutability

Python object types are either mutable or immutable. An *immutable* object cannot be altered once created, so assigning a new name to it creates a copy in memory. A *mutable* object's value may be changed, so assigning a new name to it does *not* create a copy. Therefore, if two names refer to the same mutable object, any changes to the object will be reflected in both names.

Achtung!

Making a copy of a mutable object is possible, but doing so incorrectly can cause subtle problems. For example, suppose we have a dictionary mapping items to their base prices, and we want make a similar dictionary that accounts for a small sales tax.

```
>>> 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 now 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. In the above code, we 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. Python has several methods that seem to change immutable objects. These methods actually work by making copies of objects. We can therefore determine which object types are mutable and which are immutable by using the equal sign and "changing" the objects.

```
>>> 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
```

Since altering one name altered the other, we conclude that no copy has been made and that therefore Python dictionaries are mutable. If we repeat this process with a different type and the two names are different in the end, we will know that a copy had been made and the type is immutable.

Following the example given above, determine which object types are mutable and which are immutable. Use the following operations to modify each of the given types.

numbers	num += 1
strings	word += 'a'
lists	list.append(1)
tuples	tuple += (1,)
dictionaries	dict[1] = 'a'

Print a statement of your conclusions.

Modules

A Python *module* is a file containing Python 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.

All import statements should occur at the top of the file, below the header but before any other code. Thus we expand our example of typical Python file from the previous lab to the following:

```
# filename.py
"""This is the file header.
The header contains basic information about the file.
"""

import math
import numpy as np
from scipy import linalg as la
import matplotlib.pyplot as plt

if __name__ == "__main__":
    pass                            # 'pass' is a temporary placeholder.
```

Note

The modules imported in this example are some of the most important modules for this curriculum. The NumPy, SciPy, and Matplotlib modules will be presented in detail in subsequent labs.

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

Importing Syntax

There are several ways to use the `import` statement.

1. `import <module>` makes the specified module available under the alias of its own name. For example, the `math` module has a function called `sqrt()` that computes the square root of the input.

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

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

```
>>> import math as m            # The name 'm' gives access to the math
>>> m.sqrt(2)                   # module, but the name 'math' does not.
1.4142135623730951
>>> math.sqrt(2)
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: name 'math' 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 math import sqrt       # The name 'sqrt' gives access to the
>>> sqrt(2)                     # square root function, but the rest of
1.4142135623730951              # the math module is unavailable.
>>> math.sin(2)
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
NameError: name 'math' is not defined
```

In each case, the far right 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`.

```
data = range(4)

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: [1, 2, 3, 4]
This file was executed from the command line or an interpreter.
```

Executing the file with IPython's `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: [1, 2, 3, 4]
This file was executed from the command line or an interpreter.

In [2]: display()
Data: [1, 2, 3, 4]
```

Importing the file also executes each line, 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 direcly 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: [1, 2, 3, 4]
```

Problem 3. Create a module called `calculator.py`. Write a function that returns the sum of two arguments, a function that returns the product of two arguments, and a function that returns the square root of a single argument. When the file is either run or imported, nothing should be executed.

In your main solutions file, import your new calculator module. Using only the functions defined in the module, write a new function that calculates the length of the hypotenuse of a right triangle given the lengths of the other two sides.

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`.

```
# 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. We modify the source file to correct the mistake, then run the file again in IPython:

```
# 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).

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.

Module	Description / Purpose
`csv`	Comma Separated Value (CSV) file parsing and writing.
`math`	Standard mathematical functions.
`random`	Random variable generators.
`sys`	Tools for interacting with the interpreter.
`time`	Time value generation and manipulation.
`timeit`	Measuring execution time of small code snippets.

Using IPython's object introspection, we can learn about how to use the various modules and functions in the standard library very quickly. 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' from '/anaconda/lib/python2.7/lib-dynload/math.so'>
File:        /anaconda/lib/python2.7/lib-dynload/math.so
Docstring:
This module is always available.  It 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'.
math.acos       math.atanh      math.e          math.factorial
math.hypot      math.log10      math.sin        math.acosh
math.ceil       math.erf        math.floor      math.isinf
math.log1p      math.sinh       math.asin       math.copysign
math.erfc       math.fmod       math.isnan      math.modf
math.sqrt       math.asinh      math.cos        math.exp
math.frexp      math.ldexp      math.pi         math.tan
math.atan       math.cosh       math.expm1      math.fsum
math.lgamma     math.pow        math.tanh       math.atan2
math.degrees    math.fabs       math.gamma      math.log
math.radians    math.trunc

In [4]: math.sqrt?
Type:        builtin_function_or_method
String form: <built-in function sqrt>
Docstring:
sqrt(x)

Return the square root of x.
```

The Sys Module

The `sys` (system) module includes methods for interacting with the Python interpreter. The module has a name `argv` that is a list of arguments passed to the interpreter when it runs a Python file. Consider the following simple script:

```
# echo.py
import sys

print(sys.argv)
```

If this file is run from the command line with additional arguments it will print them out. Note that command line arguments are parsed as strings.

```
$ python echo.py I am the walrus
['echo', 'I', 'am', 'the', 'walrus']
```

Command line arguments can be used to control a script's behavior, as in the following example:

```
# example3.py
"""Read a single command line argument and print it in all caps."""
import sys

if len(sys.argv) != 2:
    print("One extra command line argument is required")
else:
    print(sys.argv[1].upper())
```

Now specify the behavior of the script with command line arguments.

```
$ python example3.py
An extra command line argument is required

$ python example3.py hello
HELLO
```

In IPython, command line arguments are specified after the `run` command.

```
In [1]: run example3.py hello
HELLO
```

Another way to get input from the program user is to prompt the user for text. The built-in function `raw_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.

```
In [1]: x = raw_input("Enter a value for x: ")
Enter a value for x: 20             # Type '20' and press 'enter.'

In [2]: x
Out[2]: '20'                        # Note that x contains a string.

In [3]: y = int(raw_input("Enter an integer for y: "))
Enter an integer for y: 16          # Type '16' and press 'enter.'

In [4]: y
Out[4]: 16                          # Note that y contains an integer.
```

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.

```
>>> import random
>>> numbers = 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 random.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
```

Problem 4. *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 roll two dice. Any single set of integers from the player's remaining numbers that sum up to the sum of the dice roll may then be removed from the game. 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. See `https://www.youtube.com/watch?v=vLlZGBQ6TKs` for a visual explanation.

Modify your solutions file so that when the file is run (but **not** when it is imported), the user plays a game of shut the box. The provided module `box.py` contains some functions that will be useful in your implementation of the game. You do not need to know how the functions work, but you do need to be able to import and use them correctly.

Your game should match the following specifications:

- Before starting the game, record the player's name.
  - If the user provides a single command line argument after the filename, use that argument for their name.
  - If there are no command line arguments, use `raw_input()` to prompt the user for their name.
- Keep track of the remaining numbers.
- Use the `random` module to simulate rolling two six-sided dice. If the sum of the remaining numbers is 6 or less, role only one die.
- Print the remaining numbers and the sum of the dice roll at each turn.
- If the game is not over, prompt the user for numbers to eliminate after each dice roll. 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.
- When the game is over, print the player's name and score.

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

```
$ python solutions.py TA

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

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

Numbers left: [2, 3, 6, 7, 8, 9]
Roll: 9
Numbers to eliminate: 9

Numbers left: [2, 3, 6, 7, 8]
Roll: 8
Numbers to eliminate: 8

Numbers left: [2, 3, 6, 7]
Roll: 10
Numbers to eliminate: 3 7

Numbers left: [2, 6]
Roll: 8
Numbers to eliminate: 2 6

Score for player TA: 0 points
Congratulations!! You shut the box!
```

```
$ python solutions.py
Player name: Math TA

Numbers left: [1, 2, 3, 4, 5, 6, 7, 8, 9]
Roll: 7
Numbers to eliminate: Seven
Invalid input
Numbers to eliminate: 1, 2, 4
Invalid input
Numbers to eliminate: 10
Invalid input
Numbers to eliminate: 1 2 4

Numbers left: [3, 5, 6, 7, 8, 9]
Roll: 4
Game over!

Score for player Math TA: 38 points
```

```
$ python solutions.py
Player name: TA

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

Numbers left: [1, 2, 3, 4, 5, 6, 8, 9]
Roll: 9
Numbers to eliminate: 1 2 3 4
Invalid input
Numbers to eliminate: 1 2 6

Numbers left: [3, 4, 5, 8, 9]
Roll: 7
Numbers to eliminate: 7
Invalid input
Numbers to eliminate: 3 4

Numbers left: [5, 8, 9]
Roll: 2
Game over!

Score for player TA: 22 points
```

Additional Material

More Built-in Functions

Experiment with the following built-in functions:

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.
`input()`	Prompt the user for input and evaluate it (`eval(raw_input())`).
`map()`	Apply a function to every item of the input iterable and return a list of the results.
`reduce()`	Apply a function accepting two arguments cumulatively to the items of the input iterable from left to right (thus reducing it to a single output).

The `all()` and `any()` functions are particularly useful for creating readable code when used in conjunction with conditionals or list comprehensions.

```
>>> from random import randint
# Get 5 random numbers between 1 and 10, inclusive
>>> numbers = [randint(1,10) for _ in xrange(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 4. 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.

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, we must make the associated directories into 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` to be executable. Executing the package will run `__init__.py` and then `__main__.py`, but importing the package will only run `__init__.py`.

Suppose we are working on a project with the following file directories.

```
ssb/
        __init__.py
        __main__.py
        characters/
                __init__.py
                link.py
                mario.py
        gui/
                __init__.py
                character_select.py
                main_menu.py
                stage_select.py
        play/
                __init__.py
                damage.py
                sounds.py
        spoilers/
                __init__.py
                __main__.py
                jigglypuff.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>`. Thus the files `ssb/__init__.py` and `ssb/__main__.py` might be as follows:

```
print("ssb/__init__.py")

# Load a subpackage.
import characters
# Import the start() function from the main_menu module in the gui/ subpackage.
from gui.main_menu import start
```

```
print("ssb/__main__.py")

# Load a name that was already imported in ssb/__init__.py
from . import start
print("\tStarting game...")
start()
```

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`).

For instance, `ssb/gui/__init__.py` might need to access `ssb/` subpackages.

```
print("ssb/gui/__init__.py")

from .. import characters           # Import a subpackage from the ssb/ package.
from ..play import sounds           # Import a module from the play/ subpackage.
print("\tInitializing GUI...")
```

To execute a package, run Python from the shell with the flag `-m` (for "module-name") and exclude the extension `.py`. Suppose each file prints out its own name.

```
$ python -m ssb
ssb/__init__.py                     # First run ssb/__init__.py.
ssb/characters/__init__.py
ssb/gui/__init__.py
ssb/play/__init__.py
ssb/play/sounds.py
    Initializing GUI...
ssb/gui/main_menu.py
ssb/__main__.py                     # Then run ssb/__main__.py.
    Starting game...
```

Separate directories by a period to run a subpackage.

```
print("ssb/spoilers/__main__.py")

print("\tChallenger Approaching!")
import jigglypuff
```

```
# Run spoilers/ as a subpackage.
$ python -m ssb.spoilers
ssb/__init__.py                     # First run ssb/__init__.py.
ssb/characters/__init__.py
ssb/gui/__init__.py
ssb/play/__init__.py
ssb/play/sounds.py
    Initializing GUI...
ssb/gui/main_menu.py
ssb/spoilers/__init__.py            # Second run ssb/spoilers/__init__.py.
ssb/spoilers/__main__.py            # Finally run ssb/spoilers/__main__.py.
    Challenger Approaching!
ssb/spoilers/jigglypuff.py

# Change directories to ssb/ to run spoilers/ without running ssb/.
$ cd ssb
$ python -m spoilers
ssb/spoilers/__init__.py            # First run ssb/spoilers/__init__.py.
ssb/spoilers/__main__.py            # Then run ssb/spoilers/__main__.py.
    Challenger Approaching!
ssb/spoilers/jigglypuff.py
```

See `https://docs.python.org/2/tutorial/modules.html#packages` for more details and examples on this topic.