1 Introduction to Python

Lab Objective: *Python is a powerful general-purpose programming language. It can be used interactively, allowing for very rapid development. Python has many powerful scientific computing tools, making it an ideal language for applied and computational mathematics. In this introductory lab we introduce Python syntax, data types, functions, and control flow tools. These Python basics are an essential part of almost every problem you will solve and almost every program you will write.*

Getting Started

Python is quickly gaining momentum as a fundamental tool in scientific computing. *Anaconda* is a free distribution service by Continuum Analytics, Inc., that includes the cross-platform Python *interpreter* (the software that actually executes Python code) and many Python libraries that are commonly used for applied and computational mathematics. To install Python via Anaconda, go to `https://www.anaconda.com/download/`, download the installer for **Python 3.6** corresponding to your operating system, and follow the on-screen instructions. Python 2.7 is still popular in the scientific community (as of 2018), but more and more libraries are moving support to Python 3.

Running Python

Python files are saved with a `.py` extension. For beginners, we strongly recommend using a simple text editor for writing Python files, though many free IDEs (Integrated Development Environments—large applications that facilitate code development with some sophisticated tools) are also compatible with Python. For now, the simpler the coding environment, the better.

A plain Python file looks similar to the following code.

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

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

The # character creates a single-line *comment*. Comments are ignored by the interpreter and serve as annotations for the accompanying source code. A pair of three quotes, `""" """` or `''' '''`, creates a multi-line string literal, which may also be used as a multi-line comment. A triple-quoted string literal at the top of the file serves as the *header* for the file. The header typically identifies the author and includes instructions on using the file. Executable Python code comes after the header.

Problem 1. Open the file named `python_intro.py` (or create the file in a text editor if you don't have it). Add your information to the header at the top, then add the following code.

```
if __name__ == "__main__":
    print("Hello, world!")          # Indent with four spaces (NOT a tab).
```

Open a command prompt (*Terminal* on Linux or Mac and *Command Prompt* or *GitBash* on Windows) and navigate to the directory where the new file is saved. Use the command `ls` (or `DIR` on Windows) to list the files and folders in the current directory, `pwd` (`CD ,` on Windows) to print the working directory, and `cd` to change directories.

```
$ pwd                               # Print the working directory.
/Users/Guest
$ ls                                # List the files and folders here.
Desktop     Documents      Downloads      Pictures       Music
$ cd Documents                      # Navigate to a different folder.
$ pwd
/Users/Guest/Documents
$ ls                                # Check to see that the file is here.
python_intro.py
```

Now the Python file can be executed with the following command:

```
$ python python_intro.py
```

If `Hello, world!` is displayed on the screen, you have just successfully executed your first Python program!

IPython

Python can be run interactively using several interfaces. The most basic of these is the Python interpreter. In this and subsequent labs, the triple brackets >>> indicate that the given code is being executed one line at a time via the Python interpreter.

```
$ python                            # Start the Python interpreter.
>>> print("This is plain Python.")  # Execute some code.
This is plain Python.
```

There are, however, more useful interfaces. Chief among these is *IPython*,[1] [PG07, jup] which is included with the Anaconda distribution. To execute a script in IPython, use the `%run` command.

[1] See `https://ipython.org/` and `https://jupyter.org/`.

```
>>> exit()                          # Exit the Python interpreter.
$ ipython                           # Start IPython.

In [1]: print("This is IPython!")   # Execute some code.
This is IPython!

In [2]: %run python_intro.py         # Run a particular Python script.
Hello, world!
```

One of the biggest advantages of IPython is that it supports *object introspection*, whereas the regular Python interpreter does not. Object introspection quickly reveals all methods and attributes associated with an object. IPython also has a built-in `help()` function that provides interactive help.

```
# A list is a basic Python data structure. To see the methods associated with
# a list, type the object name (list), followed by a period, and press tab.
In [1]: list.   # Press 'tab'.
              append()  count()   insert()  remove()
              clear()   extend()  mro()     reverse()
              copy()    index()   pop()     sort()

# To learn more about a specific method, use a '?' and hit 'Enter'.
In [1]: list.append?
Signature: list.append(self, object, /)
Docstring: Append object to the end of the list.
Type:      method_descriptor

In [2]: help()                      # Start IPython's interactive help utility.

help> list                          # Get documentation on the list class.
Help on class list in module builtins:

class list(object)
 |  list(iterable=(),/)
 |  # ...                           # Press 'q' to exit the info screen.

help> quit                          # End the interactive help session.
```

Note

Use IPython side-by-side with a text editor to test syntax and small code snippets quickly. Testing small pieces of code in IPython **before** putting it into a program reveals errors and greatly speeds up the coding process. Consult the internet with questions; `stackoverflow.com` is a particularly valuable resource for answering common programming questions.

The best way to learn a new coding language is by actually writing code. Follow along with the examples in the yellow code boxes in this lab by executing them in an IPython console. Avoid copy and paste for now; your fingers need to learn the language as well.

Python Basics

Arithmetic

Python can be used as a calculator with the regular +, -, *, and / operators. Use ** for exponentiation and % for modular division.

```
>>> 3**2 + 2*5                    # Python obeys the order of operations.
19

>>> 13 % 3                        # The modulo operator % calculates the
1                                 # remainder: 13 = (3*4) + 1.
```

In most Python interpreters, the underscore character _ is a variable with the value of the previous command's output, like the ANS button on many calculators.

```
>>> 12 * 3
36
>>> _ / 4
9.0
```

Data comparisons like < and > act as expected. The == operator checks for numerical equality and the <= and >= operators correspond to $\leq$ and $\geq$, respectively. To connect multiple boolean expressions, use the operators and, or, and not.[2]

```
>>> 3 > 2.99
True
>>> 1.0 <= 1 or 2 > 3
True
>>> 7 == 7 and not 4 < 4
True

>>> True and True and True and True and True and False
False
>>> False or False or False or False or False or True
True
>>> True or not True
True
```

[2]In many other programming languages, the and, or, and not operators are written as &&, ||, and !, respectively. Python's convention is much more readable and does not require parentheses.

Variables

Variables are used to temporarily store data. A **single** equals sign = assigns one or more values (on the right) to one or more variable names (on the left). A **double** equals sign == is a comparison operator that returns True or False, as in the previous code block.

Unlike many programming languages, Python does not require a variable's data type to be specified upon initialization. Because of this, Python is called a *dynamically typed* language.

```
>>> x = 12                           # Initialize x with the integer 12.
>>> y = 2 * 6                        # Initialize y with the integer 2*6 = 12.
>>> x == y                           # Compare the two variable values.
True

>>> x, y = 2, 4                      # Give both x and y new values in one line.
>>> x == y
False
```

Functions

To define a function, use the def keyword followed by the function name, a parenthesized list of parameters, and a colon. Then indent the function body using exactly **four** spaces.

```
>>> def add(x, y):
...     return x + y                 # Indent with four spaces.
```

ACHTUNG!

Many other languages use the curly braces {} to delimit blocks, but Python uses whitespace indentation. In fact, whitespace is essentially the only thing that Python is particularly picky about compared to other languages: **mixing tabs and spaces confuses the interpreter and causes problems**. Most text editors have a setting to set the indentation type to spaces so you can use the tab key on your keyboard to insert four spaces (sometimes called *soft tabs*). For consistency, **never** use tabs; **always** use spaces.

Functions are defined with *parameters* and called with *arguments*, though the terms are often used interchangeably. Below, width and height are parameters for the function area(). The values 2 and 5 are the arguments that are passed when calling the function.

```
>>> def area(width, height):         # Define the function.
...     return width * height
...
>>> area(2, 5)                       # Call the function.
10
```

Python functions can also return multiple values.

```
>>> def arithmetic(a, b):
...     return a - b, a * b         # Separate return values with commas.
...
>>> x, y = arithmetic(5, 2)         # Unpack the returns into two variables.
>>> print(x, y)
3 10
```

The keyword `lambda` is a shortcut for creating one-line functions. For example, the polynomials $f(x) = 6x^3 + 4x^2 - x + 3$ and $g(x, y, z) = x + y^2 - z^3$ can be defined as functions in one line each.

```
# Define the polynomials the usual way using 'def'.
>>> def f(x):
...     return 6*x**3 + 4*x**2 - x + 3
>>> def g(x, y, z):
...     return x + y**2 - z**3

# Equivalently, define the polynomials quickly using 'lambda'.
>>> f = lambda x: 6*x**3 + 4*x**2 - x + 3
>>> g = lambda x, y, z: x + y**2 - z**3
```

Note

Documentation is important in every programming language. Every function should have a *docstring*—a string literal in triple quotes just under the function declaration—that describes the purpose of the function, the expected inputs and return values, and any other notes that are important to the user. Short docstrings are acceptable for very simple functions, but more complicated functions require careful and detailed explanations.

```
>>> def add(x, y):
...     """Return the sum of the two inputs."""
...     return x + y

>>> def area(width, height):
...     """Return the area of the rectangle with the specified width
...     and height.
...     """
...     return width * height
...
>>> def arithmetic(a, b):
...     """Return the difference and the product of the two inputs."""
...     return a - b, a * b
```

Lambda functions cannot have custom docstrings, so the `lambda` keyword should be only be used as a shortcut for very simple or intuitive functions that need no additional labeling.

Problem 2. The volume of a sphere with radius r is $V = \frac{4}{3}\pi r^3$. In your Python file from Problem 1, define a function called `sphere_volume()` that accepts a single parameter r. Return the volume of the sphere of radius r, using 3.14159 as an approximation for π (for now). Also write an appropriate docstring for your function.

To test your function, call it under the `if __name__ == "__main__"` clause and print the returned value. Run your file to see if your answer is what you expect it to be.

Achtung!

The `return` statement instantly ends the function call and passes the return value to the function caller. However, functions are not required to have a return statement. A function without a return statement implicitly returns the Python constant `None`, which is similar to the special value `null` of many other languages. Calling `print()` at the end of a function does **not** cause a function to return any values.

```
>>> def oops(i):
...     """Increment i (but forget to return anything)."""
...     print(i + 1)
...
>>> def increment(i):
...     """Increment i."""
...     return i + 1
...
>>> x = oops(1999)                  # x contains 'None' since oops()
2000                                # doesn't have a return statement.
>>> y = increment(1999)             # However, y contains a value.
>>> print(x, y)
None 2000
```

If you have any intention of using the results of a function, use a `return` statement.

It is also possible to specify *default values* for a function's parameters. In the following example, the function `pad()` has three parameters, and the value of `c` defaults to 0. If it is not specified in the function call, the variable `c` will contain the value 0 when the function is executed.

```
>>> def pad(a, b, c=0):
...     """Print the arguments, plus an zero if c is not specified."""
...     print(a, b, c)
...
>>> pad(1, 2, 3)                # Specify each parameter.
1 2 3
>>> pad(1, 2)                   # Specify only non-default parameters.
1 2 0
```

Arguments are passed to functions based on position or name, and positional arguments must be defined before named arguments. For example, `a` and `b` must come before `c` in the function definition of `pad()`. Examine the following code blocks demonstrating how positional and named arguments are used to call a function.

```
# Try defining printer with a named argument before a positional argument.
>>> def pad(c=0, a, b):
...     print(a, b, c)
...
SyntaxError: non-default argument follows default argument
```

```
# Correctly define pad() with the named argument after positional arguments.
>>> def pad(a, b, c=0):
...     """Print the arguments, plus an zero if c is not specified."""
...     print(a, b, c)
...

# Call pad() with 3 positional arguments.
>>> pad(2, 4, 6)
2 4 6

# Call pad() with 3 named arguments. Note the change in order.
>>> pad(b=3, c=5, a=7)
7 3 5

# Call pad() with 2 named arguments, excluding c.
>>> pad(b=1, a=2)
2 1 0

# Call pad() with 1 positional argument and 2 named arguments.
>>> pad(1, c=2, b=3)
1 3 2
```

Problem 3. The built-in `print()` function has the useful keyword arguments `sep` and `end`. It accepts any number of positional arguments and prints them out with `sep` inserted between values (defaulting to a space), then prints `end` (defaulting to the *newline character* `'\n'`).

Write a function called `isolate()` that accepts five arguments. The function should print the first three arguments separated by 5 spaces and then print the last two arguments with a single space seperating the last three arguments. For example,

```
>>> isolate(1, 2, 3, 4, 5)
1     2     3 4 5
```

Achtung!

In previous versions of Python, print() was a *statement* (like return), not a function, and could therefore be executed without parentheses. However, it lacked keyword arguments like sep and end. If you are using Python 2.7, include the following line at the top of the file to turn the print statement into the new print() function.

```
>>> from __future__ import print_function
```

Data Types and Structures

Numerical Types

Python has four numerical data types: int, long, float, and complex. Each stores a different kind of number. The built-in function type() identifies an object's data type.

```
>>> type(3)                        # Numbers without periods are integers.
int

>>> type(3.0)                      # Floats have periods (3. is also a float).
float
```

Python has two types of division: integer and float. The / operator performs float division (true fractional division), and the // operator performs integer division, which rounds the result down to the next integer. If both operands for // are integers, the result will be an int. If one or both operands are floats, the result will be a float. Regular division with / always returns a float.

```
>>> 15 / 4                         # Float division performs as expected.
3.75
>>> 15 // 4                        # Integer division rounds the result down.
3
>>> 15. // 4
3.0
```

Achtung!

In previous versions of Python, using / with two integers performed integer division, even in cases where the division was not even. This can result in some incredibly subtle and frustrating errors. If you are using Python 2.7, always include a . on the operands or cast at least one as a float when you want float division.

```
# PYTHON 2.7
>>> 15 / 4                         # The answer should be 3.75, but the
3                                  # interpreter does integer division!
```

```
>>> 15. / float(4)                   # 15. and float(4) are both floats, so
3.75                                 # the interpreter does float division.
```

Alternatively, including the following line at the top of the file redefines the / and // operators so they are handled the same way as in Python 3.

```
>>> from __future__ import division
```

Python also supports complex numbers computations by pairing two numbers as the real and imaginary parts. Use the letter j, not i, for the imaginary part.

```
>>> x = complex(2,3)                 # Create a complex number this way...
>>> y = 4 + 5j                       # ...or this way, using j (not i).
>>> x.real                           # Access the real part of x.
2.0
>>> y.imag                           # Access the imaginary part of y.
5.0
```

Strings

In Python, strings are created with either single or double quotes. To concatenate two or more strings, use the + operator between string variables or literals.

```
>>> str1 = "Hello"
>>> str2 = 'world'
>>> my_string = str1 + " " + str2 + '!'
>>> my_string
'Hello world!'
```

Parts of a string can be accessed using *slicing*, indicated by square brackets `[ ]`. Slicing syntax is `[start:stop:step]`. The parameters `start` and `stop` default to the beginning and end of the string, respectively. The parameter `step` defaults to 1.

```
>>> my_string = "Hello world!"
>>> my_string[4]                 # Indexing begins at 0.
'o'
>>> my_string[-1]                # Negative indices count backward from the end.
'!'

# Slice from the 0th to the 5th character (not including the 5th character).
>>> my_string[:5]
'Hello'

# Slice from the 6th character to the end.
>>> my_string[6:]
```

```
'world!'

# Slice from the 3rd to the 8th character (not including the 8th character).
>>> my_string[3:8]
'lo wo'

# Get every other character in the string.
>>> my_string[::2]
'Hlowrd'
```

Problem 4. Write two new functions, called `first_half()` and `backward()`.

1. `first_half()` should accept a parameter and return the first half of it, excluding the middle character if there is an odd number of characters.
   (Hint: the built-in function `len()` returns the length of the input.)

2. The `backward()` function should accept a parameter and reverse the order of its characters using slicing, then return the reversed parameter.
   (Hint: The `step` parameter used in slicing can be negative.)

Use IPython to quickly test your syntax for each function.

Lists

A Python `list` is created by enclosing comma-separated values with square brackets `[ ]`. Entries of a list do **not** have to be of the same type. Access entries in a list with the same indexing or slicing operations used with strings.

```
>>> my_list = ["Hello", 93.8, "world", 10]
>>> my_list[0]
'Hello'
>>> my_list[-2]
'world'
>>> my_list[:2]
['Hello', 93.8]
```

Common list methods (functions) include `append()`, `insert()`, `remove()`, and `pop()`. Consult IPython for details on each of these methods using object introspection.

```
>>> my_list = [1, 2]                # Create a simple list of two integers.
>>> my_list.append(4)               # Append the integer 4 to the end.
>>> my_list.insert(2, 3)            # Insert 3 at location 2.
>>> my_list
[1, 2, 3, 4]
>>> my_list.remove(3)               # Remove 3 from the list.
>>> my_list.pop()                   # Remove (and return) the last entry.
4
```

```
>>> my_list
[1, 2]
```

Slicing is also very useful for replacing values in a list.

```
>>> my_list = [10, 20, 30, 40, 50]
>>> my_list[0] = -1
>>> my_list[3:] = [8, 9]
>>> print(my_list)
[-1, 20, 30, 8, 9]
```

The `in` operator quickly checks if a given value is in a list (or another iterable, including strings).

```
>>> my_list = [1, 2, 3, 4, 5]
>>> 2 in my_list
True
>>> 6 in my_list
False
>>> 'a' in "xylophone"              # 'in' also works on strings.
False
```

Tuples

A Python `tuple` is an ordered collection of elements, created by enclosing comma-separated values with parentheses `(` and `)`. Tuples are similar to lists, but they are much more rigid, have less built-in operations, and cannot be altered after creation. Lists are therefore preferable for managing dynamic ordered collections of objects.

When multiple objects are returned by a function, they are returned as a tuple. For example, recall that the `arithmetic()` function returns two values.

```
>>> x, y = arithmetic(5,2)                      # Get each value individually,
>>> print(x, y)
3 10
>>> both = arithmetic(5,2)                      # or get them both as a tuple.
>>> print(both)
(3, 10)
```

Problem 5. Write a function called `list_ops()`. Define a list with the entries `"bear"`, `"ant"`, `"cat"`, and `"dog"`, in that order. Then perform the following operations on the list:

1. Append `"eagle"`.
2. Replace the entry at index 2 with `"fox"`.
3. Remove (or pop) the entry at index 1.
4. Sort the list in reverse alphabetical order.

5. Replace `"eagle"` with `"hawk"`.
   (Hint: the list's `index()` method may be helpful.)

6. Add the string `"hunter"` to the last entry in the list.

Return the resulting list of strings.

Work out (on paper) what the result should be, then check that your function returns the correct list. Consider printing the list at each step to see the intermediate results.

Sets

A Python `set` is an unordered collection of distinct objects. Objects can be added to or removed from a set after its creation. Initialize a set with curly braces `{ }`, separating the values by commas, or use `set()` to create an empty set. Like mathematical sets, Python sets have operations like union, intersection, difference, and symmetric difference.

```
# Initialize some sets. Note that repeats are not added.
>>> gym_members = {"Doe, John", "Doe, John", "Smith, Jane", "Brown, Bob"}
>>> print(gym_members)
{'Doe, John', 'Brown, Bob', 'Smith, Jane'}

>>> gym_members.add("Lytle, Josh")      # Add an object to the set.
>>> gym_members.discard("Doe, John")    # Delete an object from the set.
>>> print(gym_members)
{'Lytle, Josh', 'Brown, Bob', 'Smith, Jane'}

>>> gym_members.intersection({"Lytle, Josh", "Henriksen, Ian", "Webb, Jared"})
{'Lytle, Josh'}
>>> gym_members.difference({"Brown, Bob", "Sharp, Sarah"})
{'Lytle, Josh', 'Smith, Jane'}
```

Dictionaries

Like a set, a Python `dict` (dictionary) is an unordered data type. A dictionary stores key-value pairs, called *items*. The values of a dictionary are indexed by its keys. Dictionaries are initialized with curly braces, colons, and commas. Use `dict()` or `{}` to create an empty dictionary.

```
>>> my_dictionary = {"business": 4121, "math": 2061, "visual arts": 7321}
>>> print(my_dictionary["math"])
2061

# Add a value indexed by 'science' and delete the 'business' keypair.
>>> my_dictionary["science"] = 6284
>>> my_dictionary.pop("business")       # Use 'pop' or 'popitem' to remove.
4121
>>> print(my_dictionary)
{'math': 2061, 'visual arts': 7321, 'science': 6284}
```

```
# Display the keys and values.
>>> my_dictionary.keys()
dict_keys(['math', 'visual arts', 'science'])
>>> my_dictionary.values()
dict_values([2061, 7321, 6284])
```

As far as data access goes, lists are like dictionaries whose keys are the integers $0, 1, \ldots, n-1$, where n is the number of items in the list. The keys of a dictionary need not be integers, but they must be *immutable*, which means that they must be objects that cannot be modified after creation. We will discuss mutability more thoroughly in the Standard Library lab.

Type Casting

The names of each of Python's data types can be used as functions to cast a value as that type. This is particularly useful for converting between integers and floats.

```
# Cast numerical values as different kinds of numerical values.
>>> x = int(3.0)
>>> y = float(3)
>>> z = complex(3)
>>> print(x, y, z)
3 3.0 (3+0j)

# Cast a list as a set and vice versa.
>>> set([1, 2, 3, 4, 4])
{1, 2, 3, 4}
>>> list({'a', 'a', 'b', 'b', 'c'})
['a', 'c', 'b']

# Cast other objects as strings.
>>> str(['a', str(1), 'b', float(2)])
"['a', '1', 'b', 2.0]"
>>> str(list(set([complex(float(3))])))
'[(3+0j)]'
```

Control Flow Tools

Control flow blocks dictate the order in which code is executed. Python supports the usual control flow statements including `if` statements, `while` loops and `for` loops.

The If Statement

An `if` statement executes the indented code **if** (and only if) the given condition holds. The `elif` statement is short for "else if" and can be used multiple times following an if statement, or not at all. The `else` keyword may be used at most once at the end of a series of `if`/`elif` statements.

```
>>> food = "bagel"
>>> if food == "apple":                 # As with functions, the colon denotes
...     print("72 calories")            # the start of each code block.
... elif food == "banana" or food == "carrot":
...     print("105 calories")
... else:
...     print("calorie count unavailable")
...
calorie count unavailable
```

Problem 6. Write a function called `pig_latin()`. Accept a string parameter `word`, translate it into Pig Latin, then return the translation. Specifically, if `word` starts with a vowel, add “hay” to the end; if `word` starts with a consonant, take the first character of `word`, move it to the end, and add “ay”.
(Hint: use the `in` operator to check if the first letter is a vowel.)

The While Loop

A `while` loop executes an indented block of code **while** the given condition holds.

```
>>> i = 0
>>> while i < 10:
...     print(i, end=' ')               # Print a space instead of a newline.
...     i += 1                          # Shortcut syntax for i = i+1.
...
0 1 2 3 4 5 6 7 8 9
```

There are two additional useful statements to use inside of loops:

1. `break` manually exits the loop, regardless of which iteration the loop is on or if the termination condition is met.
2. `continue` skips the current iteration and returns to the top of the loop block if the termination condition is still not met.

```
>>> i = 0
>>> while True:
...    print(i, end=' ')
...    i += 1
...    if i >= 10:
...        break                        # Exit the loop.
...
0 1 2 3 4 5 6 7 8 9

>>> i = 0
>>> while i < 10:
...     i += 1
...     if i % 3 == 0:
...         continue                # Skip multiples of 3.
...     print(i, end=' ')
1 2 4 5 7 8 10
```

The For Loop

A `for` loop iterates over the items in any *iterable*. Iterables include (but are not limited to) strings, lists, sets, and dictionaries.

```
>>> colors = ["red", "green", "blue", "yellow"]
>>> for entry in colors:
...    print(entry + "!")
...
red!
green!
blue!
yellow!
```

The `break` and `continue` statements also work in for loops, but a `continue` in a for loop will automatically increment the index or item, whereas a `continue` in a while loop makes no automatic changes to any variable.

```
>>> for word in ["It", "definitely", "looks", "pretty", "bad", "today"]:
...     if word == "definitely":
...         continue
...     elif word == "bad":
...         break
...     print(word, end=' ')
...
It looks pretty
```

In addition, Python has some very useful built-in functions that can be used in conjunction with the `for` statement:

1. `range(start, stop, step)`: Produces a sequence of integers, following slicing syntax. If only one argument is specified, it produces a sequence of integers from 0 to the argument, incrementing by one. This function is used **very** often.

2. `zip()`: Joins multiple sequences so they can be iterated over simultaneously.

3. `enumerate()`: Yields both a count and a value from the sequence. Typically used to get both the index of an item and the actual item simultaneously.

4. `reversed()`: Reverses the order of the iteration.

5. `sorted()`: Returns a new list of sorted items that can then be used for iteration.

Each of these functions except for `sorted()` returns an *iterator*, an object that is built specifically for looping but not for creating actual lists. To put the items of the sequence in a collection, use `list()`, `set()`, or `tuple()`.

```
# Strings and lists are both iterables.
>>> vowels = "aeiou"
>>> colors = ["red", "yellow", "white", "blue", "purple"]

# Iterate by index.
>>> for i in range(5):
...     print(i, vowels[i], colors[i])
...
0 a red
1 e yellow
2 i white
3 o blue
4 u purple

# Iterate through both sequences at once.
>>> for letter, word in zip(vowels, colors):
...     print(letter, word)
...
a red
e yellow
i white
o blue
u purple

# Get the index and the item simultaneously.
>>> for i, color in enumerate(colors):  #
...     print(i, color)
...
0 red
1 yellow
2 white
3 blue
4 purple

# Iterate through the list in sorted (alphabetical) order.
>>> for item in sorted(colors):
...     print(item, end=' ')
...
blue purple red white yellow

# Iterate through the list backward.
>>> for item in reversed(colors):
...     print(item, end=' ')
...
purple blue white yellow red

# range() arguments follow slicing syntax.
>>> list(range(10))                 # Integers from 0 to 10, exclusive.
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> list(range(4, 8))              # Integers from 4 to 8, exclusive.
[4, 5, 6, 7]

>>> set(range(2, 20, 3))           # Every third integer from 2 to 20.
{2, 5, 8, 11, 14, 17}
```

Problem 7. This problem originates from `https://projecteuler.net`, an excellent resource for math-related coding problems.

A palindromic number reads the same both ways. The largest palindrome made from the product of two 2-digit numbers is $9009 = 91 \times 99$. Write a function called `palindrome()` that finds and returns the largest palindromic number made from the product of two 3-digit numbers.

List Comprehension

A *list comprehension* uses for loop syntax between square brackets to create a list. This is a powerful, efficient way to build lists. The code is concise and runs quickly.

```
>>> [float(n) for n in range(5)]
[0.0, 1.0, 2.0, 3.0, 4.0]
```

List comprehensions can be thought of as "inverted loops", meaning that the body of the loop comes before the looping condition. The following loop and list comprehension produce the same list, but the list comprehension takes only about two-thirds the time to execute.

```
>>> loop_output = []
>>> for i in range(5):
...     loop_output.append(i**2)
...
>>> list_output = [i**2 for i in range(5)]
```

Tuple, set, and dictionary comprehensions can be done in the same way as list comprehensions by using the appropriate style of brackets on the end.

```
>>> colors = ["red", "blue", "yellow"]
>>> {c[0]:c for c in colors}
{'y': 'yellow', 'r': 'red', 'b': 'blue'}

>>> {"bright " + c for c in colors}
{'bright blue', 'bright red', 'bright yellow'}
```

Problem 8. The alternating harmonic series is defined as follows.

$$\sum_{n=1}^{\infty} \frac{(-1)^{(n+1)}}{n} = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \ldots = \ln(2)$$

Write a function called `alt_harmonic()` that accepts an integer n. Use a list comprehension to quickly compute the first n terms of this series (be careful not to compute only $n-1$ terms). The sum of the first 500,000 terms of this series approximates $\ln(2)$ to five decimal places. (Hint: consider using Python's built-in `sum()` function.)

Additional Material

Further Reading

Refer back to this and other introductory labs often as you continue getting used to Python syntax and data types. As you continue your study of Python, we strongly recommend the following readings.

- The official Python tutorial: `http://docs.python.org/3.6/tutorial/introduction.html` (especially chapters 3, 4, and 5).
- Section 1.2 of the SciPy lecture notes: `http://scipy-lectures.github.io/`.
- PEP8 - Python style guide: `http://www.python.org/dev/peps/pep-0008/`.

Generalized Function Input

On rare occasion, it is necessary to define a function without knowing exactly what the parameters will be like or how many there will be. This is usually done by defining the function with the parameters `*args` and `**kwargs`. Here `*args` is a list of the positional arguments and `**kwargs` is a dictionary mapping the keywords to their argument. This is the most general form of a function definition.

```
>>> def report(*args, **kwargs):
...     for i, arg in enumerate(args):
...         print("Argument " + str(i) + ":", arg)
...     for key in kwargs:
...         print("Keyword", key, "-->", kwargs[key])
...
>>> report("TK", 421, exceptional=False, missing=True)
Argument 0: TK
Argument 1: 421
Keyword missing --> True
Keyword exceptional --> False
```

See `https://docs.python.org/3.6/tutorial/controlflow.html` for more on this topic.

Function Decorators

A *function decorator* is a special function that “wraps” other functions. It takes in a function as input and returns a new function that pre-processes the inputs or post-processes the outputs of the original function.

```
>>> def typewriter(func):
...     """Decorator for printing the type of output a function returns"""
...     def wrapper(*args, **kwargs):
...         output = func(*args, **kwargs)      # Call the decorated function.
...         print("output type:", type(output)) # Process before finishing.
...         return output                       # Return the function output.
...     return wrapper
```

The outer function, `typewriter()`, returns the new function `wrapper()`. Since `wrapper()` accepts `*args` and `**kwargs` as arguments, the input function `func()` could accepts any number of positional or keyword arguments.

Apply a decorator to a function by tagging the function's definition with an @ symbol and the decorator name.

```
>>> @typewriter
... def combine(a, b, c):
...     return a*b // c
```

Placing the tag above the definition is equivalent to adding the following line of code after the function definition:

```
>>> combine = typewriter(combine)
```

Now calling `combine()` actually calls `wrapper()`, which then calls the original `combine()`.

```
>>> combine(3, 4, 6)
output type: <class 'int'>
2
>>> combine(3.0, 4, 6)
output type: <class 'float'>
2.0
```

Function decorators can also customized with arguments. This requires another level of nesting: the outermost function must define and return a decorator that defines and returns a wrapper.

```
>>> def repeat(times):
...     """Decorator for calling a function several times."""
...     def decorator(func):
...         def wrapper(*args, **kwargs):
...             for _ in range(times):
...                 output = func(*args, **kwargs)
...             return output
...         return wrapper
...     return decorator
...
>>> @repeat(3)
... def hello_world():
...     print("Hello, world!")
...
>>> hello_world()
Hello, world!
Hello, world!
Hello, world!
```

See `https://www.python.org/dev/peps/pep-0318/` for more details.